Großer Beleg

Reengineering des OCL2-Parsers

bearbeitet von
Nils Thieme

Version 1.1
Änderungsdatum 17.02.2008

Technische Universität Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Betreuerin: Dr.-Ing. Birgit Demuth
Hochschullehrer: Prof. Dr. rer. nat. habil. Uwe Aßmann

Eingereicht am 30. November 2007

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 30. November 2007

einer wissenschaftlichen Ausgabe, einem nachgelassenen Werk, einem Lichtbild, einer Datenbank, einem Tonträger, einer Funksendung, einem Laufbild oder einer Darbietung eines ausübenden Künstlers.

i) Unter dem "Werk" wird eine persönliche geistige Schöpfung verstanden, die Ihnen unter den Bedingungen dieser Lizenz angeboten wird.

2. Schranken des Urheberrechts. Diese Lizenz lässt sämtliche Befugnisse unberührt, die sich aus den Schranken des Urheberrechts,aus dem Erschöpfungsgrundsatz oder anderen Beschränkungen der Ausschließlichkeitsrechte des Rechtsinhabers ergeben.

3. Lizenzierung. Unter den Bedingungen dieses Lizenzvertrages räumt Ihnen der Lizenzgeber ein lizenzgebührenfreies, räumlich und zeitlich (für die Dauer des Urheberrechts oder verwandten Schutzrechts) unbeschränktes einfaches Nutzungsrecht ein, den Schutzgegenstand in der folgenden Art und Weise zu nutzen:

   a) den Schutzgegenstand in körperlicher Form zu verwerten, insbesondere zu vervielfältigen, zu verbreiten und auszustellen;
   b) den Schutzgegenstand in unkörperlicher Form öffentlich wiederzugeben, insbesondere vorzutragen, aufzuführen und vorzuführen, öffentlich zugänglich zu machen, zu senden, durch Bild- und Tonträger wiederzugeben sowie Funksendungen und öffentliche Zugänglichmachungen wiederzugeben;
   c) den Schutzgegenstand auf Bild- oder Tonträger aufzunehmen, Lichtbilder davon herzustellen, weiterzusenden und in dem in a. und b. genannten Umfang zu verwerten;
   d) den Schutzgegenstand zu bearbeiten oder in anderer Weise umzugestalten und die Bearbeitungen zu veröffentlichen und in dem in a. bis c. genannten Umfang zu verwerten;

   Die genannten Nutzungsrechte können für alle bekannten Nutzungsarten ausgeübt werden. Die genannten Nutzungsrechte beinhalten das Recht, solche Veränderungen an dem Werk vorzunehmen, die technisch erforderlich sind, um die Nutzungsrechte für alle Nutzungsarten wahrzunehmen. Insbesondere sind davon die Anpassung an andere Medien und auf andere Dateiformate umfasst.

4. Beschränkungen. Die Einräumung der Nutzungsrechte gemäß Ziffer 3 erfolgt ausdrücklich nur unter den folgenden Bedingungen:

   a) Sie dürfen den Schutzgegenstand ausschließlich unter den Bedingungen dieser Lizenz vervielfältigen, verbreiten oder öffentlich wiedergeben, und Sie müssen stets eine Kopie oder die vollständige Internetadresse in Form des Uniform-Resource-Identifier (URI) dieser Lizenz beifügen, wenn Sie den Schutzgegenstandvervielfältigen, verbreiten oder öffentlich wiedergeben. Sie dürfen keine Vertragsbedingungen anbieten oder fordern, die die Bedingungen dieser Lizenz oder die durch sie gewährten Rechte ändern oder beschränken. Sie dürfen den Schutzgegenstand nicht unterlizenzieren. Sie müssen alle Hinweise unverändert lassen, die auf diese Lizenz und den Haftungsausschluss hinweisen. Sie dürfen den Schutzgegenstand mit keinen technischen Schutzmaßnahmen versehen, die den Zugang oder den Gebrauch des Schutzgegenstandes in einer Weise kontrollieren, die mit den Bedingungen dieser Lizenz im Widerspruch stehen. Die genannten Beschränkungen gelten auch für den Fall, dass der Schutzgegenstand einen Bestandteil eines Sammelwerkes bildet; sie

verlangen aber nicht, dass das Sammelwerk insgesamt zum Gegenstand dieser Lizenz gemacht wird. Wenn Sie ein Sammelwerk erstellen, müssen Sie - soweit dies praktikabel ist - auf die Mitteilung eines Lizenzgebers oder Urhebers hin aus dem Sammelwerk jeglichen Hinweis auf diesen Lizenzgeber oder diesen Urheber entfernen. Wenn Sie den Schutzgegenstand bearbeiten, müssen Sie - soweit dies praktikabel ist- auf die Aufforderung eines Rechtsinhabers hin von der Bearbeitung jeglichen Hinweis auf diesen Rechtsinhaber entfernen.

b) Sie dürfen eine Bearbeitung ausschließlich unter den Bedingungen dieser Lizenz, einer späteren Version dieser Lizenz mit denselben Lizenzelementen wie diese Lizenz oder einer Creative Commons iCommons Lizenz, die dieselben Lizenzelemente wie diese Lizenz enthält (z.B. Namensnennung - Nicht-kommerziell - Weitergabe unter gleichen Bedingungen 2.0 Japan), vervielfältigen, verbreiten oder öffentlich wiedergeben. Sie müssen stets eine Kopie oder die Internetadresse in Form des Uniform-Resource-Identifier (URI) dieser Lizenz oder einer anderen Lizenz der im vorhergehenden Satz beschriebenen Art beifügen, wenn Sie die Bearbeitung vervielfältigen, verbreiten oder öffentlich wiedergeben. Sie dürfen keine Vertragsbedingungen anbieten oder fordern, die die Bedingungen dieser Lizenz oder die durch sie gewährten Rechte ändern oder beschränken, und Sie müssen alle Hinweise unverändert lassen, die auf diese Lizenz und den Haftungsausschluss hinweisen. Sie dürfen eine Bearbeitung nicht mit technischen Schutzmaßnahmen versehen, die den Zugang oder den Gebrauch der Bearbeitung in einer Weise kontrollieren, die mit den Bedingungen dieser Lizenz im Widerspruch stehen. Die genannten Beschränkungen gelten auch für eine Bearbeitung als Bestandteil eines Sammelwerkes; sie erfordern aber nicht, dass das Sammelwerk insgesamt zum Gegenstand dieser Lizenz gemacht wird.

c) Sie dürfen die in Ziffer 3 gewährten Nutzungsrechte in keiner Weise verwenden, die hauptsächlich auf einen geschäftlichen Vorteil oder eine vertraglich geschuldete geldwerte Vergütung abzielt oder darauf gerichtet ist. Erhalten Sie im Zusammenhang mit der Einräumung der Nutzungsrechte ebenfalls einen Schutzgegenstand, ohne dass eine vertragliche Verpflichtung hierzu besteht, so wird dies nicht als geschäftlicher Vorteil oder vertraglich geschuldete geldwerte Vergütung angesehen, wenn keine Zahlung oder geldwerte Vergütung in Verbindung mit dem Austausch der Schutzgegenstände geleistet wird (z.B. File-Sharing).

d) Wenn Sie den Schutzgegenstand oder eine Bearbeitung oder ein Sammelwerk vervielfältigen, verbreiten oder öffentlich wiedergeben, müssen Sie alle Urhebervermerke für den Schutzgegenstand unverändert lassen und die Urheberschaft oder Rechtsinhaberschaft in einer der von Ihnen vorgenommenen Nutzung angemessenen Form anerkennen, indem Sie den Namen (oder das Pseudonym, falls ein solches verwendet wird) des Urhebers oder Rechteinhabers nennen, wenn dieser angegeben ist. Dies gilt auch für den Titel des Schutzgegenstandes, wenn dieser angeben ist, sowie - in einem vernünftigerweise durchführbaren Umfang - für die mit dem Schutzgegenstand zu verbindende Internetadresse in Form des Uniform-Resource-Identifier (URI), wie sie der Lizenzgeber angegeben hat, sofern dies geschehen ist, es sei denn, diese Internetadresse verweist nicht auf den Urhebervermerk oder die Lizenzinformationen zu dem Schutzgegenstand. Bei einer Bearbeitung ist ein

Hinweis darauf aufzuführen, in welcher Form der Schutzgegenstand in die Bearbeitung eingegangen ist (z.B. “Französische Übersetzung des ... (Werk) durch ... (Urheber)“ oder “Das Drehbuch beruht auf dem Werk des ... (Urheber)“). Ein solcher Hinweis kann in jeder angemessenen Weise erfolgen, wobei jedoch bei einer Bearbeitung, einer Datenbank oder einem Sammelwerk der Hinweis zumindest an gleicher Stelle und in ebenso auffälliger Weise zu erfolgen hat wie vergleichbare Hinweise auf andere Rechtsinhaber.

e) Obwohl die gemäss Ziffer 3 gewährten Nutzungsrechte in umfassender Weise ausgeübt werden dürfen, findet diese Erlaubnis ihre gesetzliche Grenze in den Persönlichkeitsrechten der Urheber und ausübenden Künstler, deren berechtigte geistige und persönliche Interessen bzw. deren Ansehen oder Ruf nicht dadurch gefährdet werden dürfen, dass ein Schutzgegenstand über das gesetzlich zulässige Maß hinaus beeinträchtigt wird.

5. Gewährleistung. Sofern dies von den Vertragsparteien nicht anderweitig schriftlich vereinbart,, bietet der Lizenzgeber keine Gewährleistung für die erteilten Rechte, außer für den Fall, dass Mängel arglistig verschwiegen wurden. Für Mängel anderer Art, insbesondere bei der mangelhaften Lieferung von Verkörperungen des Schutzgegenstandes, richtet sich die Gewährleistung nach der Regelung, die die Person, die Ihnen den Schutzgegenstand zur Verfügung stellt, mit Ihnen außerhalb dieser Lizenz vereinbart, oder - wenn eine solche Regelung nicht getroffen wurde - nach den gesetzlichen Vorschriften.

6. Haftung. Über die in Ziffer 5 genannte Gewährleistung hinaus haftet Ihnen der Lizenzgeber nur für Vorsatz und grobe Fahrlässigkeit.

7. Vertragsende

   a) Dieser Lizenzvertrag und die durch ihn eingeräumten Nutzungsrechte enden automatisch bei jeder Verletzung der Vertragsbedingungen durch Sie. Für natürliche und juristische Personen, die von Ihnen eine Bearbeitung, eine Datenbank oder ein Sammelwerk unter diesen Lizenzbedingungen erhalten haben, gilt die Lizenz jedoch weiter, vorausgesetzt, diese natürlichen oder juristischen Personen erfüllen sämtliche Vertragsbedingungen. Die Ziffern 1, 2, 5, 6, 7 und 8 gelten bei einer Vertragsbeendigung fort.

   b) Unter den oben genannten Bedingungen erfolgt die Lizenz auf unbegrenzte Zeit (für die Dauer des Schutzrechts). Dennoch behält sich der Lizenzgeber das Recht vor, den Schutzgegenstand unter anderen Lizenzbedingungen zu nutzen oder die eigene Weitergabe des Schutzgegenstandes jederzeit zu beenden, vorausgesetzt, dass solche Handlungen nicht dem Widerruf dieser Lizenz dienen (oder jeder anderen Lizenzierung, die auf Grundlage dieser Lizenz erfolgt ist oder erfolgen muss) und diese Lizenz wirksam bleibt, bis Sie unter den oben genannten Voraussetzungen endet.

8. Schlussbestimmungen

   a) Jedes Mal, wenn Sie den Schutzgegenstand vervielfältigen, verbreiten oder öffentlich wiedergeben, bietet der Lizenzgeber dem Erwerber eine Lizenz für den Schutzgegenstand unter denselben Vertragsbedingungen an, unter denen er Ihnen die Lizenz eingeräumt hat.

   b) Jedes Mal, wenn Sie eine Bearbeitung vervielfältigen, verbreiten oder öffentlich wiedergeben, bietet der Lizenzgeber dem Erwerber eine Lizenz für den

ursprünglichen Schutzgegenstand unter denselben Vertragsbedingungen an, unter denen er Ihnen die Lizenz eingeräumt hat.

c) Sollte eine Bestimmung dieses Lizenzvertrages unwirksam sein, so wird die Wirksamkeit der übrigen Lizenzbestimmungen dadurch nicht berührt, und an die Stelle der unwirksamen Bestimmung tritt eine Ersatzregelung, die dem mit der unwirksamen Bestimmung angestrebten Zweck am nächsten kommt.

d) Nichts soll dahingehend ausgelegt werden, dass auf eine Bestimmung dieses Lizenzvertrages verzichtet oder einer Vertragsverletzung zugestimmt wird, so lange ein solcher Verzicht oder eine solche Zustimmung nicht schriftlich vorliegen und von der verzichtenden oder zustimmenden Vertragspartei unterschrieben sind

e) Dieser Lizenzvertrag stellt die vollständige Vereinbarung zwischen den Vertragsparteien hinsichtlich des Schutzgegenstandes dar. Es gibt keine weiteren ergänzenden Vereinbarungen oder mündlichen Abreden im Hinblick auf den Schutzgegenstand. Der Lizenzgeber ist an keine zusätzlichen Abreden gebunden, die aus irgendeiner Absprache mit Ihnen entstehen könnten. Der Lizenzvertrag kann nicht ohne eine übereinstimmende schriftliche Vereinbarung zwischen dem Lizenzgeber und Ihnen abgeändert werden.

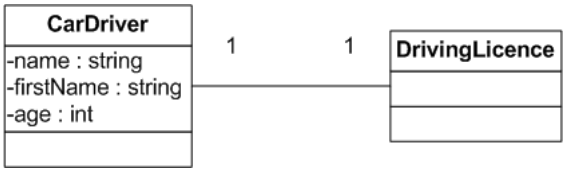
f) Auf diesen Lizenzvertrag findet das Recht der Bundesrepublik Deutschland Anwendung.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In der heutigen Zeit wird Software unter zu Hilfenahme von Modellen entwickelt. Dabei spielt die UML (*Unified Modeling Language*) [UML] eine wichtige Rolle. Mit der UML ist es aber nur beschränkt möglich, ein Softwaresystem vollständig zu beschreiben. Das Problem soll am folgenden Beispiel veranschaulicht werden:



Dieses kleine Modell drückt aus, dass ein Autofahrer einen Führerschein besitzen muss (1-1-Assoziation). Mit der UML ist es aber nicht möglich, eine Bedingung oder eine Einschränkung (im Englischen *Constraint* genannt) einzufügen. Im Beispielmodell wäre es sinnvoll, die Bedingung einzufügen, dass nur volljährige Personen einen Führerschein besitzen dürfen. An diesem Punkt setzt die Sprache OCL (*Object Constraint Language*) an, die 1995 von der OMG (*Object Management Group*) [OMG] ins Leben gerufen wurde. Mit Hilfe der OCL ist es möglich, einem UML-Diagramm Bedingungen hinzuzufügen. Für das Autofahrerbeispiel könnte eine Bedingung wie folgt aussehen:

```
context CarDriver
        inv: age >= 18
```

Dadurch wird das Modell vollständiger beschrieben.

OCL ist für die vollständige Beschreibung von Modellen sehr wichtig. Aus diesem Grund hat sich der Lehrstuhl für Softwaretechnologie der Fakultät Informatik (TU-Dresden) zum Ziel gesetzt den Standard bekannter zu machen. Um den Standard Leben einzuhauchen, wurde 1999 das Dresdner OCL-Toolkit im Rahmen des Großen Beleges [FIN] entwickelt. In dieser Werkzeugsammlung dreht sich alles um OCL. Verschiedene Module zeigen, wie OCL eingesetzt werden kann (Erzeugung von Java-Quelltext, der die Bedingungen während der Ausführung eines Programms überwacht; ein SQL-Generator; ein Interpreter, der ein Modell ad hoc ausführen kann). Alle diese Module benötigen einen Parser, der die textuellen Ausdrücke in eine für den Computer verständliche Form bringt. Im alten Dresdner OCL-Toolkit konnte der Parser nicht alle Ausdrücke, die mit OCL beschrieben werden können, verarbeiten. Das lag teilweise an der Infrastruktur, auf die der Parser zugriff. Ein Ziel dieser Arbeit war es, den Parser möglichst vollständig zu implementieren und den OCL-Standard [OCL2] einzuhalten.

Das Dresdner OCL-Toolkit erfuhr durch den Beleg [Braeuer] einen großen Fortschritt. Zum einen wurde es auf die Eclipse Plattform migriert und zum anderen wurde das sog. *Pivotmodell* eingeführt. Um das Pivotmodell zu verstehen, sollte die sog. *Metamodellierung* verstanden werden. Einen guten Einstieg in dieses Thema gibt beispielsweise [SVEH].

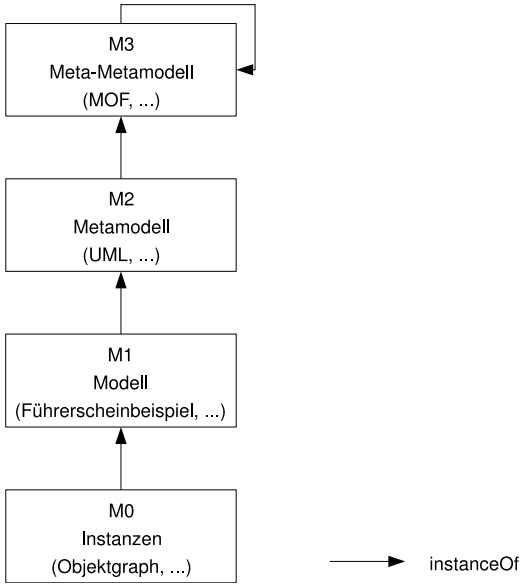


Abbildung 1.1: Hierarchie der Metamodellierung

In Abbildung 1.1 ist die Hierarchie der Metamodellierung abgebildet. Das Pivotmodell befindet sich auf der Stufe M2 der Metamodelle. Ziel ist es, OCL-Ausdrücke auf das Pivotmodell zu beziehen anstatt auf das UML-Metamodell. Zu diesem Zweck muss der alte OCL-Parser angepasst werden, da dieser eine Verbindung zum UML-Metamodell und zum Meta-Metamodell MOF (*Meta Object Facility*) herstellt.

Das Pivotmodell dient als Austauschformat zwischen Metamodellen. Das bedeutet, dass eine Transformation zwischen dem UML-Metamodell und dem Pivotmodell spezifiziert werden kann, aber auch andere Metamodelle können in das Pivotmodell transformiert werden (siehe [BRA]). Der Vorteil dieser Vorgehensweise liegt in der einheitlichen Integration aller Module des Dresdner OCL-Toolkits. Alle Module, einschließlich des Parsers, basieren auf dem Pivotmodell (zur Zeit dieser Arbeit wurden noch nicht alle Module angepasst). Das heißt, die Module müssen nicht mehr für jedes Metamodell neu angepasst werden. Es reicht, wenn eine Abbildung zwischen diesem Metamodell und dem Pivotmodell hergestellt wird. Weitere Informationen zum Pivotmodell und zur Transformation können in [Braeuer] nachgelesen werden.

1.2 Vorgehen

Ausgangspunkt für die Neuimplementierung des OCL-Parsers bildete das Pivotmodell (siehe [Braeuer]). Daher schien es sinnvoll, sich zunächst mit diesem auseinanderzusetzen.

Des Weiteren wurde das Dresdner OCL-Toolkit in Eclipse-PlugIn's aufgeteilt, so dass es notwendig wurde, diese Architektur zu verstehen, um den Parser darin einzubetten.

In [ANS] wurde der alte OCL-Parser implementiert. Aus dieser Diplomarbeit ging eine Erweiterung des Parser-Generators *SableCC* hervor, mit dessen Hilfe der alte Parser implementiert wurde. Es lag nahe, diese Erweiterung auch für den neuen Parser zu

nutzen. Dazu musste eine Einarbeitung in das angepasste Werkzeug erfolgen.

Um ein Gefühl für die Arbeit mit der SableCC-Erweiterung zu bekommen, wurde die Sprache PL0 implementiert. Dies hatte den Vorteil, den Parser-Generator zu evaluieren, ohne zu viel Aufwand zu betreiben, da PL0 eine sehr kleine Sprache mit beschränktem Funktionsumfang ist.

In eine engere Auswahl trat auch das Werkzeug *JastAdd*. Mit diesem wurde ebenfalls die Sprache PL0 implementiert.

Die gewonnenen Erfahrungen aus dem Umgang mit beiden Werkzeugen flossen in die Implementierung des neuen OCL-Parsers ein.

1.3 Vergleich mit anderen OCL-Parsern

In letzter Zeit wurden viele verschiedene Werkzeuge implementiert, die mit OCL arbeiten. So gibt es Werkzeuge, die OCL mit der MDA [MDA] verbinden, um Metamodelle mit semantischen Regeln zu annotieren. Andere Werkzeuge generieren aus OCL-Ausdrücken Quelltext, der während der Ausführung eines Programms prüft, ob die spezifizierten OCL-Ausdrücke eingehalten werden.

Alle diese Werkzeuge benötigen einen OCL-Parser, um die textuelle Eingabe eines OCL-Ausdruckes weiterverarbeiten zu können. In diesem Abschnitt werden vier OCL-Parser vorgestellt.

In diesem Abschnitt werden die Begriffe *abstrakter Syntaxbaum* und *abstrakter Syntaxgraph* verwendet. Diese werden im Kapitel 2 auf Seite 19 näher vorgestellt.

1.3.1 KentOCL

Der OCL-Parser des KentOCL-Werkzeugs [KEN] wurde mit dem Parser-Generator *CUP* [CUP] und dem Lexer-Generator *JFlex* [JFlex] generiert. Es wird eine abstrakte Syntax und ein Metamodell für OCL erzeugt (mit dem Werkzeug *KMF* [KFM]). Instanzen der abstrakten Syntax sind abstrakte Syntaxbäume, während die Instanzen des Metamodells abstrakte Syntaxgraphen darstellen. Der von CUP generierte Parser erzeugt aus einem OCL-Ausdruck zunächst einen abstrakten Syntaxbaum. Anschließend wird dieser mit Anwendung der semantischen Regeln in einen abstrakten Syntaxgraphen transformiert. Diese Transformation ist über das Besuchermuster [GOF] implementiert. Da die Besucherklasse sämtliche semantischen Regeln auswertet, ist diese Klasse besonders komplex.

1.3.2 EclipseOCL

Im Rahmen der *Modeling Development Tools* [EMF] des Eclipse Projektes [Eclipse] gibt es eine OCL-Implementierung. Der OCL-Parser basiert auf einem generierten Parser, der von dem sehr flexiblen Werkzeug *LPG* [LPG] erzeugt wurde. Dieser Parser erzeugt aus einem Textausdruck einen abstrakten Syntaxbaum, der anschließend in einen abstrakten Syntaxgraphen transformiert wird. Die letzte Transformation wird durch eine abgespeckte Besucherklasse aus dem Besuchermuster durchgeführt, die die semantischen Regeln prüft.

Sowohl der abstrakte Syntaxbaum als auch der abstrakte Syntaxgraph sind Instanzen einer abstrakten Syntax und eines Metamodells. Beides wird durch das *Eclipse Modeling Framework* [EMF] generiert.

OCL-Parser	benutzt. Werkzeuge	Phasen	Besuchermuster
KentOCL	CUP, JFlex	2	ja
EclipseOCL	LPG	2	ja
Naomi	SableCC	1	ja
bestehender OCL-Parser	ExtSableCC	1	ja

Tabelle 1.1: Vergleich der OCL-Parser

1.3.3 Naomi

Der OCL-Parser des Naomi-Werkzeugs [NAO] wurde mit dem Parser-Generator *SableCC* [GAG] implementiert. Es gibt keine Übersetzung in einen abstrakten Syntaxbaum oder -graphen. Statt dessen wird der konkrete Syntaxbaum während der Auswertung der semantischen Regeln mit Typen annotiert. Da SableCC ein Framework generiert, das auf dem Besuchermuster basiert, ist auch hier die Implementierung in einer großen Klasse untergebracht.

1.3.4 Bestehender OCL-Parser

Der bestehende OCL-Parser wurde mit einer angepassten Version des Werkzeuges SableCC geschrieben (siehe [Kon]). Mit diesem Werkzeug wird ein Parser generiert, der sowohl den konkreten Syntaxbaum aufbaut als auch den abstrakten Syntaxgraphen. Die SableCC-Erweiterung generiert ein Framework, in das die semantischen Prüfungen einbettet werden. Das Framework basiert auf dem Besuchermuster und so werden alle semantischen Regeln in einer einzigen großen Datei untergebracht, die schwer verständlich ist. Zu dem ist der OCL-Parser vollständig abhängig von der SableCC-Erweiterung. Ein Austausch wird sehr schwierig.

1.3.5 Vergleich der OCL-Parser

Die drei Werkzeuge werden in Tabelle 1.1 gegenübergestellt.

Alle vier Werkzeuge implementieren die semantischen Prüfungen mit Hilfe des Besuchermusters. Dies führt meist zu einer einzelnen großen Klasse, die unübersichtlich ist und eine Einarbeitung erschwert. Die Werkzeuge *KentOCL* und *EclipseOCL* bauen auf einen Parser auf, der in zwei Phasen arbeitet. Dadurch wird eine gewisse Unabhängigkeit von den verwendeten Werkzeugen erreicht (siehe Abschnitt 7.2 auf Seite 129). Dieses Prinzip wird im neuen OCL-Parser übernommen.

1.4 Struktur der Kapitel

In diesem Abschnitt wird der Inhalt der einzelnen Kapitel kurz erläutert.

Kapitel Grundlagen In dieser Arbeit wird ein neuer OCL-Parser implementiert. Um die Arbeitsweise eines Parsers zu verstehen, werden in diesem Kapitel die Grundlagen vermittelt. Zunächst wird die *lexikalische Analyse* erläutert, die einzelne Zeichen zu größeren Einheiten zusammenfasst. Der nächste Abschnitt erläutert die *syntaktische Analyse*, die einen Quelltext zu einer Baumstruktur umformt. Um diese Umformung zu verstehen, werden einige theoretische Grundlagen gelegt. Anschließend wird der Begriff

der *Attributierung* eingeführt, der erläutert, wie mit einem Quelltext Semantik verbunden werden kann. Zum Schluss wird der Begriff der *abstrakten Syntax* eingeführt, da dieser für die Implementierung des OCL-Parsers sehr wichtig ist.

Kapitel PL0 Vor der Implementierung des OCL-Parsers musste ein geeignetes Werkzeug gefunden werden, das diese Arbeit unterstützt. Um mehrere Werkzeuge evaluieren zu können, wurde die Sprache PL0 (eine Sprache ähnlich zu Pascal) mit unterschiedlichen Werkzeugen implementiert. In diesem Kapitel wird die Sprache durch Morphemklassen, einer kontextfreien Grammatik und einer abstrakten Syntax vollständig beschrieben. Zum Schluss werden semantische Regeln formal eingeführt.

Kapitel SableCC Ein Werkzeug, was in die nähere Auswahl trat, war *SableCC*.
SableCC ist ein Parser-Generator, der keinen Quelltext in der Spezifikation erlaubt, sondern eine Framework generiert. Der OCL-Parser wurde im Rahmen der Diplomarbeit von [ANS] mit einer angepassten Version dieses Parser-Generator implementiert. Diese wird ebenfalls vorgestellt.

Kapitel JastAdd JastAdd ist ein relativ junges Werkzeug, das Attributauswertungen unterstützt. Es wird zunächst vorgestellt. Um einen einfachen Einblick in die Arbeit mit diesem Werkzeug zu geben, wird beschrieben, wie sich die Sprache PL0 damit umsetzen lässt. Dies ist Vorbereitung für das Verständnis des OCL-Parsers.

Kapitel OCL-Parser In diesem Kapitel wird der OCL-Parser vorgestellt, der das Hauptanliegen dieser Arbeit ist. Es wird auf die tatsächlich verwendeten Werkzeuge eingegangen und wie der Parser entstanden ist. Außerdem werden die semantischen Prüfungen vorgestellt, die mit der Sprache OCL verbunden sind.

Kapitel Abschließende Betrachtung Dieses Kapitel gibt eine Zusammenfassung der gesamten Arbeit und führt eine Bewertung des Ergebnisses durch. Auch behandelt es kurz das Thema der Erweiterbarkeit des neuen OCL-Parsers.

2 Grundlagen

In diesem Kapitel sollen die Grundlagen für den Bau eines Parsers vermittelt werden. Die einzelnen Zeichen eines Quelltextes werden zunächst in größere Einheiten zusammengefasst. Dies wird im ersten Abschnitt besprochen. Anschließend kann aus diesen größeren Einheiten ein Baum aufgebaut werden, was der Abschnitt der syntaktischen Analyse bearbeitet. Semantik spielt eine wesentliche Rolle in der Verarbeitung von Quelltexten, daher wird im nächsten Abschnitt auf die Attributierung eingegangen. Zum Schluss wird die sog. abstrakte Syntax behandelt, von der ausführlich im OCL-Parser Gebrauch gemacht wird.

2.1 Lexikalische Analyse

Die lexikalische Analyse bildet die ersten Phase des Analysevorgangs. In diesem Abschnitt soll sie näher vorgestellt werden. Zunächst wird erläutert, welche Aufgabe die lexikalische Analyse besitzt. Anschließend werden ein paar theoretische Grundlagen behandelt. Im darauffolgenden Abschnitt werden typische Kategorien, in die Lexeme eingeteilt werden können, vorgestellt. Am Schluss wird kurz auf die Werkzeugunterstützung eingegangen.

2.1.1 Prozess der lexikalischen Analyse

Die lexikalische Analyse findet in der ersten Phase des Analysevorgangs statt. Sie besitzt die Aufgabe einen Strom von Zeichen in einen Strom von *Morphemen* (im Englischen *Token* genannt) zu transformieren. Ein Morphem besteht aus zwei Teilen: einer Zeichenkette - auch *Lexem* genannt - und einer Kategorie. Die lexikalische Analyse muss zum einen Zeichen, zu Lexemen gruppieren, und zum anderen, diese Lexeme einer Kategorie zuordnen. Dies wird mit endlichen deterministischen Automaten realisiert, dessen Grundlagen mit den sprachlichen Konstrukten im nächsten Abschnitt erläutert werden.

2.1.2 Theoretische Grundlagen

Definition 1 (Wort) *Sei* $\Sigma = \{a, b, c, ...\}$ *eine Menge von Symbolen, auch Alphabet genannt. Ein Wort* w *über* Σ *ist eine Zeichenkette, deren Symbole Elemente aus* Σ *sind* $w = a_0a_1a_2...a_n$*, wobei* $a_i \in \Sigma$ *gilt. Ein besonderes Wort ist das leere Wort* ϵ*.*

Definition 2 (Länge eines Wortes) *Die Länge eines Wortes wird mit* $\mid w \mid$ *angegeben, wobei gilt:* $w = a_1a_2a_3...a_n, \mid w \mid = n$ *und* $\mid \epsilon \mid = 0$*.*

Definition 3 (Konkatenation zweier Worte) *Seien* $w_1 = a_0a_1a_2...a_n$ *und* $w_2 = b_0b_1b_2...b_r$ *zwei Worte, wobei gilt* $a_i, b_i \in \Sigma$*. Dann bezeichnet*

$$w_1 \circ w_2 = a_0a_1a_2...a_nb_0b_1b_2...b_n$$

die Konkatenation von w_1 *und* w_2*. Oft wird das Konkatenationszeichen* $\circ$ *weggelassen.*

Definition 4 (Sternoperator für Mengen) *Sei* $\Sigma = \{a, b, c, ...\}$ *eine Menge von Symbolen, dann bezeichnet*

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$$

den Stern über der Menge Σ*. Dabei gilt:* $\Sigma^n = \{a_0...a_n \mid a_i \in \Sigma\}$*. Weiterhin soll gelten:* $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$*.*

Definition 5 (Sprache) *Sei* Σ^* *eine Menge von Worten. Eine Sprache* L *ist eine Teilmenge von* Σ^*

$$L \subseteq \Sigma^*$$

Definition 6 (Grammatik) *Eine Grammatik ist ein Tupel* $G = (N, \Sigma, P, S)$*, wobei*

- N *eine endliche Menge von Nichtterminalen*
- Σ *eine Menge von Symbolen (auch Nichtterminale genannt)* $(N \cap \Sigma = \emptyset)$
- P *eine endliche Menge* $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- S *ein Startsymbol* $S \in N$

bezeichnen.

Die Menge P *enthält Produktionen der Form* $A \to B$*, wobei* $A \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$ *und* $B \in (N \cup \Sigma)^*$ *gilt.*

Definition 7 (reguläre Grammatik) *Eine Grammatik heißt regulär, wenn die Menge* P *der Produktionen folgender Einschränkung unterliegt*

$$P \subseteq N \times (\{\epsilon\} \cup \Sigma \cup \Sigma N)$$

Definition 8 (endlicher deterministischer Automat) *Ein endlicher deterministischer Automat ist ein Tupel* $M = (Q, \Sigma, \delta, q_0, F)$ *wobei*

- Q *eine Menge von Zuständen*
- Σ *eine Menge von Symbolen*
- δ *eine partielle Funktion, die sog. Übergangsfunktion* $\delta : Q \times \Sigma \rightharpoonup Q$
- q_0 *einen Startzustand*
- F *eine Menge von Endzuständen*

bezeichnen.

Definition 9 (Sprache eines Automaten und einer Grammatik) *Die Sprache, die von einer Grammatik* G *erzeugt wird, wird mit*

$$\ell(G)$$

bezeichnet.

Die Sprache, die von dem endlichen deterministischen Automat M *erkannt wird, wird mit*

$$\ell(M)$$

bezeichnet.

Satz 1 *Zu jedem endlichen deterministischen Automaten M gibt es eine reguläre Grammatik G mit*

$$\ell(M) = \ell(G)$$

Satz 1 drückt aus, dass jeder endliche deterministische Automat Worte einer regulären Sprache akzeptiert. Morpheme können durch reguläre Grammatiken beschrieben und durch endliche deterministische Automaten erkannt werden. Eine reguläre Grammatik kann Worte erzeugen, in der Terminologie der lexikalische Analyse *Lexeme* genannt. Mit dieser Grammatik kann nun eine Kategorie verbunden werden, so dass allen Lexemen, die durch die Grammatik erzeugt werden können, genau eine Kategorie zugeordnet wird. Für die endlichen deterministischen Automaten trifft Ähnliches zu. Sie erkennen eine bestimmte Art von Worten/Lexemen, so dass einem Automaten eine Kategorie zugeordnet werden kann. Endliche deterministische Automaten können von einem regulären Ausdruck konstruiert werden. Da reguläre Ausdrücke in der Spezifikation von Lexer-Generatoren eine große Rolle spielen, werden diese hier definiert:

Definition 10 (Reguläre Ausdrücke, entnommen aus [ASB], Seite 252) *Sei Σ eine endliche Menge von Symbolen. Die Menge der regulären Ausdrücke über Σ ist durch folgende induktive Definition gegeben:*

1. *$\emptyset$ und ϵ sind reguläre Ausdrücke*
2. *Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck*
3. *Mit α und β sind auch $(\alpha\beta), (\alpha + \beta)$ und (α^*) reguläre Ausdrücke.*
4. *Nichts sonst ist ein regulärer Ausdruck.*

Damit wurde bisher nur die Syntax der regulären Ausdrücke definiert. Die Definition der Semantik folgt nun:

Definition 11 (Semantik Regulärer Ausdrücke, entnommen aus [ASB], Seite 252) *Die zu einem regulären Ausdruck α gehörende Sprache $\ell(\alpha)$ ist wie folgt definiert:*

$$\begin{array}{ll} \ell(\emptyset) = \emptyset & \ell(\epsilon) = \{\epsilon\} \\ \ell(a) = \{a\} & \ell(\alpha\beta) = \ell(\alpha) \circ \ell(\beta) \\ \ell(\alpha + \beta) = \ell(\alpha) \cup \ell(\beta) & \ell(\alpha^*) = \ell(\alpha)^* \end{array}$$

Die meisten Werkzeuge unterstützen eine ähnliche Syntax, um Morpheme spezifizieren zu können.

2.1.3 Lexikalische Kategorien

Wie in den vorhergehenden Abschnitten beschrieben, bekommen Lexeme eine Kategorie zugeordnet, so dass aus ihnen Morpheme werden. Typische Kategorien sind Schlüsselwörter (wie `context`, `let`, `in`, ...), Begrenzer (wie `{`, `}`, `(`, `)`, `::`, ...), Identifizierer (wie `Person`, `age`, ...), Literale (wie `5`, `'Hello world'`, ...) und nichtsichtbare Zeichen (wie Leerzeichen, Tabulatoren, ...). Für die Schlüsselwörter ist es günstig für jedes genau eine Kategorie zu nutzen (siehe Abschnitt 2.2).

2.1.4 Werkzeugunterstützung

Für die Erstellung eines Lexers gibt es eine Vielzahl von Werkzeugen, die unterschiedliche Programmiersprachen unterstützen. Das Werkzeug *JFlex* [JFlex] beispielsweise generiert einen Lexer in der Programmiersprache Java, der Lexer-Generator *Lex* [Lex] dagegen in C.

Die Lexer-Generatoren erwarten als Eingabe eine Spezifikation, in der die Morpheme mit regulären Ausdrücken definiert werden. Aus dieser Spezifikation wird ein Automat (oder mehrere) generiert. Der generierte Lexer kann dann einen Eingabestrom von Zeichen in einen Strom von Morphemen transformieren.

Bei der Spezifikation der Morpheme spielt die Reihenfolge oft eine Rolle, da mit dem *Prinzip der längsten Kette* gearbeitet wird. Dieses besagt, dass so viele Zeichen wie möglich eingelesen werden, um ein Morphem zu erkennen, selbst wenn bereits ein Morphem erkannt werden könnte. Das ist sinnvoll, um Identifizierer, die den Präfix eines Schlüsselwortes tragen als Identifizierer zu erkennen. So ist es beispielsweise möglich, den Identifizierer `contextA` als solchen zu erkennen, ohne dass die Morphemfolge `context` (Schlüsselwort) und `A` (Identifizierer) ausgegeben wird. Die Morpheme, die innerhalb der Spezifikation oben spezifiziert werden, erhalten eine höhere Priorität als die darunter liegenden. Das bedeutet, dass die Schlüsselworte möglichst oben spezifiziert werden sollten, da sie sonst als Identifizierer erkannt werden.

2.1.5 Zusammenfassung

Die lexikalische Analyse stellt die erste Phase im Prozess der Analyse dar. Sie verwandelt einen Strom von Eingabezeichen in einen Strom von Morphemen. Außerdem kann sie nichtsichtbare Zeichen wie Leerzeichen und Tabulatoren entfernen, wenn sie nicht gebraucht werden. Zu jedem Morphem kann zusätzlich die Zeilen- und Spaltennummer gespeichert werden, was nützlich für die weitere Fehlerbehandlung ist.

Die lexikalische Analyse wird mit Hilfe eines Lexers durchgeführt. Dieser wird oft aus regulären Ausdrücken generiert, die die Grundlagen für einen Automaten sind.

2.2 Syntaktische Analyse

Die syntaktische Analyse schließt sich der lexikalischen Analyse an. Sie wird durch den sog. *Parser* durchgeführt. Ein Parser nimmt einen Strom von Morphemen entgegen und transformiert diesen in den *konkreten Syntaxbaum* (siehe Abbildung 2.1).

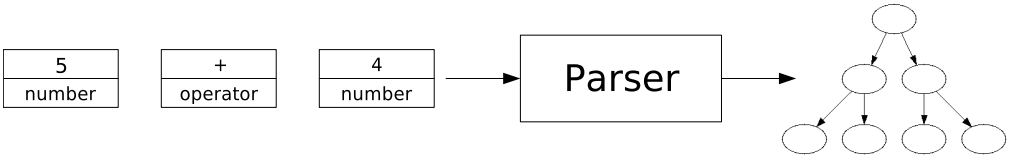


Abbildung 2.1: Aufgabe des Parsers

Der Hauptunterschied zwischen der lexikalischen und syntaktischen Analyse liegt in der Sprachverarbeitung. Während die lexiaklische Analyse mit regulären Sprachen arbeitet, werden in der syntaktischen Analyse kontfreie Sprachen verwendet.

Dieser Abschnitt gliedert sich wie folgt: zunächst werden die theoretischen Grundlagen besprochen, die für das weitere Verständnis notwenig sind. Anschließend werden allgemein die Parserverfahren vorgestellt und etwas vertiefend auf die Top-Down- und Bottom-Up-Analyse eingegangen. Da nicht jede Grammatik für einen Parser in Frage kommt, gibt der darauf folgende Abschnitt einen Überblick über die Eigenschaften von kontextfreien Grammatiken. Im letzten Abschnitt wird auf die Werkzeugunterstützung eingegangen.

2.2.1 Theoretische Grundlagen

Wie im vorhergehenden Abschnitt angedeutet, verwendet die syntaktische Analyse kontextfreie Grammatiken. Kontextfreie Grammatiken sind in der Lage kontextfreie Sprachen zu erzeugen. Die kontextfreien Grammatiken sind folgendermaßen definiert:

Definition 12 (Kontextfreie Grammatik) *Eine Grammatik $G = (N, \Sigma, P, S)$ heißt kontextfrei, wenn die Menge P der Produktionen folgender Einschränkung unterliegt*

$$P \subseteq N \times (N \cup \Sigma)^*$$

Das bedeutet, dass alle Regeln auf der linken Seite genau ein Nichtterminal und auf der rechten Seite eine beliebige Kombination aus Nichtterminalen und Terminalen (einschließlich dem leeren Wort ϵ) besitzen darf.

Der Unterschied zwischen regulären und kontextfreien Sprachen besteht darin, dass nun Worte gebildet werden können, die zwei Symbole besitzen, deren Anzahl übereinstimmt. Beispielsweise können die Worte $a^n b^n$ mit $n \in \mathbb{N}$ gebildet werden. In vielen textuellen Beschreibungssprachen, wozu auch OCL gehört, kommen Konstrukte vor, die eine solche Wortstruktur aufweisen. Man denke an die öffnenden und schließenden Klammern in aritmethischen Ausdrücken.

In der syntaktischen Analyse spielt der konkrete Syntaxbaum eine wesentliche Rolle. An dieser Stelle soll er definiert werden.

Definition 13 (Konkreter Syntaxbaum) *Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik. Ein konkreter Syntaxbaum $T = (A, E)$ ist ein zusammenhängender Baum, der folgende Eigenschaften besitzt:*

- $A = N \cup \Sigma$
- $E \subseteq A \times A$
- *es gibt einen ausgezeichneten Knoten $w \in A$, auch Wurzel genannt, für den gilt* $\forall a \in A.(a, w) \notin E \wedge w = S$
- $\forall b \in \Sigma, c \in A.(b, c) \notin E$
- $\forall a, b, c \in A.((a, b) \in E \wedge (c, b) \in E) \rightarrow a = c$

Informell gesprochen besteht ein konkreter Syntaxbaum aus den Grammatiksymbolen (Nichtterminale und Terminale). Das Startsymbol der kontextfreien Grammatik bildet die Wurzel, die Terminale die Blätter. Jeder Knoten, bis auf die Wurzel, hat genau einen Vorgänger.

Die Morpheme aus der lexikalischen Analyse werden zu den Terminalsymbolen der kontextfreien Grammatik. Das heißt, dass hier keine einzelnen Zeichen betrachtet werden müssen, sondern nur vollständige Zeichenketten. Dies erleichtert an dieser Stelle die Spezifikation der Sprache und hilft den konkreten Syntaxbaum effizienter aufzubauen.

Im weiteren Verlauf spielen die Begriffe *Linksableitung* und *Rechtsableitung* eine Rolle. Bevor diese definiert werden können, muss der Begriff der *Ableitung* eingeführt werden.

Definition 14 (Ableitung) *Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik. Dann bezeichnet*

$$xAz \Rightarrow xyz$$

eine Ableitung, wobei $x, y, z \in (N \cup \Sigma)^*$ und $A \in N$ bezeichnen, falls es eine Regel mit $A \rightarrow y$ gibt.

Eine Ableitung stellt eine Anwendung einer Grammatikregel dar. Ein Nichtterminal innerhalb einer Ableitungskette kann durch die rechte Seite einer Regel ersetzt werden, die das Nichtterminal auf der linken Seite hat.

Definition 15 (Links-/Rechtsableitung, angelehnt an [ASB], Seite 280) *Sei $G = (N, \Sigma, P, S)$ eine konextfreie Grammatik. Für die Linksableitungsrelation $\Rightarrow_L$ gilt*

$$u \Rightarrow_L u'$$

falls $u = xAz$, wobei $x \in \Sigma^*, A \in N$ und $z \in (N \cup \Sigma)^*$ und $u' = xyz$, für eine Regel $A \rightarrow y$ von G.

Sei $G = (N, \Sigma, P, S)$ eine konextfreie Grammatik. Für die Rechtsableitungsrelation $\Rightarrow_R$ gilt

$$u \Rightarrow_R u'$$

falls $u = xAz$, wobei $z \in \Sigma^*, A \in N$ und $x \in (N \cup \Sigma)^*$ und $u' = xyz$, für eine Regel $A \rightarrow y$ von G.

Informell ersetzt eine Linksableitung immer das Nichtterminal, was sich ganz links in der Ableitungskette befindet, während die Rechtsableitung immer das Nichtterminal ganz rechts ersetzt.

Die vorgestellten Parser-Verfahren beruhen auf der Idee eines *Kellerautomaten*. Dieser wird hier definiert.

Definition 16 (Kellerautomat, angelehnt an [ASB], Seite 300) *Ein endlicher deterministischer Kellerautomat ist ein Tupel $K = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$ bestehend aus*

- *einer endlichen Menge von Zuständen Q,*
- *einem Eingabealphabet Σ,*
- *einem Kelleralphabet Γ,*
- *einem Anfangszustand $q_0 \in Q$,*
- *einem Kellerstartsymbol $\#$,*
- *einer Menge $F \subseteq Q$ von Endzuständen,*
- *einer Übergangsfunktion $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$*

Im nächsten Abschnitt werden kurz zwei Parser-Verfahren vorgestellt.

2.2.2 Parserverfahren

Wie in der Einleitung beschrieben, nimmt ein Parser einen Strom von Morphemen entgegen (in diesem Abschnitt und im Folgenden wird der Begriff *Satz* verwendet, um die Eingabe des Parsers zu beschreiben) und baut daraus den konkreten Syntaxbaum auf (siehe Abbildung 2.1). Der Parser kann aus einer kontextfreien Grammatik generiert werden (siehe Abbildung 2.2).

Der zu verarbeitende Satz steht auf dem Eingabeband, das einen Zeiger hat, der auf das aktuelle Symbol zeigt. Dieser Zeiger wird während der Verarbeitung nach rechts geschoben.

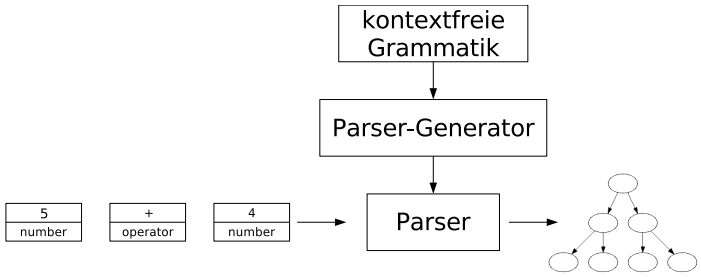


Abbildung 2.2: Parser-Generator

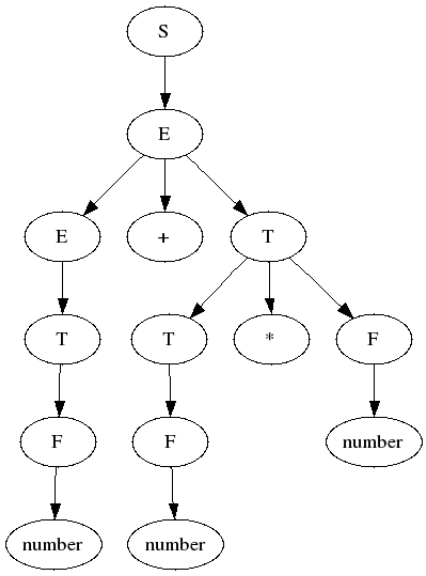
Es gibt im Wesentlichen zwei Arten von Parsern: die Top-Down- und die Bottom-Up-Parser. Ein Top-Down-Parser liest die Morpheme Stück für Stück ein und baut einen konkreten Syntaxbaum von der Wurzel zu den Blättern auf. Der Bottom-Up-Parser dagegen baut einen konkreten Syntaxbaum in umgekehrter Reihenfolge, von den Blättern zur Wurzel, auf.

Der konkrete Syntaxbaum strukturiert den vorgelegten Satz. Dazu wird folgender beispielhafte Satz betrachtet: $4 + 5 * 7$. Durch die Prioritäten der Operanden zerfällt dieser Ausdruck in die zwei Teilausdrücke 4+ und $5 * 7$. Um diesen Ausdruck in einen konkreten Syntaxbaum zu transformieren, wird eine Grammatik benötigt. Diese ist in Abbildung 2.3 dargestellt.

$$
\begin{array}{lcl}
\mathrm{S} & \rightarrow & \mathrm{E} \\
\mathrm{E} & \rightarrow & \mathrm{E} + \mathrm{T} \\
\mathrm{E} & \rightarrow & \mathrm{T} \\
\mathrm{T} & \rightarrow & \mathrm{T} * \mathrm{F} \\
\mathrm{T} & \rightarrow & \mathrm{F} \\
\mathrm{F} & \rightarrow & \mathrm{number} \\
\mathrm{F} & \rightarrow & (\mathrm{E})
\end{array}
$$

Abbildung 2.3: Grammatik einfacher arithmetischer Ausdrücke mit Linksrekursion

Die Zeichen `(`, `)` und `number` stehen für Morpheme. Für den Ausdruck $4 + 5 * 7$ und die Grammatik in Abbildung 2.3 ergibt sich der konkrete Syntaxbaum:



Für die Erzeugung eines Parser gibt es verschiedene Verfahren, die nicht für jede Parser-Art verwendet werden können.

Top-Down-Parser

Ein Top-Down-Parser erzeugt eine Linksableitung, um den vorgelegten Satz zu erkennen. Der konkrete Syntaxbaum wird dabei von links nach rechts ähnlich einer Tiefensuche aufgebaut (zuerst werden die Knoten links erzeugt, dann die Knoten rechts). Der Parser beginnt bei den Regeln, die das Startsymbol auf der linken Seite haben. Die Aufgabe des Parsers besteht nun darin, diejenige Regel auszuwählen, die das Symbol erzeugen kann, auf welches der Zeiger des Eingabebandes steht. Dieses Verhalten soll an einem kleinen Beispiel erläutert werden. Betrachtet wird die folgende Grammatik:

S	$\rightarrow$	TE
E	$\rightarrow$	+TE
E	$\rightarrow$	ϵ
T	$\rightarrow$	FT'
T'	$\rightarrow$	$*$FT'
T'	$\rightarrow$	ϵ
F	$\rightarrow$	number

Diese Grammatik erzeugt einfache arithmethische Ausdrücken. Das Symbol `number` steht dabei für das Terminal, das alle Zahlen ausdrückt.

Wenn auf dem Eingabeband die Zeichenketten $4 + 5 * 7$ und der Zeiger des Eingabebandes auf der Zahl 4 steht, dann muss es möglich sein, vom Symbol `S` aus eine Zahl zu erzeugen. Das ist tatsächlich über die Linksableitung

$$S \rightarrow TE \rightarrow FT'E \rightarrow numberT'E$$

möglich. Man sagt auch, dass `number` ein Zielsymbol der Regel $S \rightarrow TE$ ist. Weil das auf dem Eingabeband markierte Zeichen mit einem der Zielsymbole der eben betrachtenden Regel übereinstimmt, wird genau diese Regel ausgewählt. Für jede Entscheidung muss immer genau eine Regel zur Auswahl stehen, das macht das Verfahren determinitisch.

Ein Top-Down-Parser arbeitet nach dem Prinzip des Kellerautomaten, das heißt, dem Parser steht ein Keller zur Verfügung, auf dem beliebige Nichtterminale und Terminale abgelegt werden können. Über diesen Keller merkt er sich, welche Regeln er schon bearbeitet hat. Am Anfang steht nur das Startsymbol auf dem Keller. In dem Beispiel hat sich herausgestellt, dass die Regel $S \rightarrow TE$ angewendet werden kann. Das bedeutet, dass das oben liegende Symbol durch die Zeichenkette TE ersetzt wird. Das Symbol T steht dabei oben. Dann werden die Regeln betrachtet, die das Symbol T auf der linken Seite haben. Die Zielsymbolmengen werden angeschaut und es stellt sich heraus, dass das Symbol `number` (der Zeiger des Eingabebandes wurde noch nicht verrückt) in der Zielsymbolmenge der Regel $T \rightarrow FT'$ liegt. Somit wird das oberste Kellersymbol heruntergenommen und die Zeichenkette FT' auf den Keller abgelegt. Das wird so lange fortgesetzt, bis ein Terminalsymbol das oberstes Kellersymbol bildet. Wenn dieses mit dem markierten Symbol auf dem Eingabeband übereinstimmt, wird es heruntergenommen und der Zeiger des Eingabebandes wird um eines nach rechts verschoben. Wenn es nicht übereinstimmt, liegt ein Fehler vor. Erkannt wird der Satz dann, wenn kein Symbol auf dem Keller liegt und das Eingabeband leer ist. Während dieses gesamten Vorgangs wird der konkrete Syntaxbaum aufgebaut. Er ergibt sich über Bewegungen auf dem Keller.

Um eindeutig eine Regel auswählen zu können, müssen die Zielsymbolmenge jeder einzelnen Regel berechnet werden. Das geschieht mit den sog. *First-* und *Follow*-Mengen. Die Definition der beiden Menge sowie der Zielsymbolmenge kann beispielsweise in [AHO] Seite 220-222 nachgelesen werden.

Eine Anmerkung bezüglich der eingehenden kontextfreien Grammatik bleibt zu erläutern. Für jede Regel wird die Menge der Zielsymbole berechnet. Wenn es zwei Regeln gibt, die auf der linken Seite dasselbe Nichtterminal haben, und sich die Zielsymbolmengen der beiden Regeln nicht unterscheiden, dann kann diese Grammatik für das Top-Down-Verfahren nicht genutzt werden. Dazu ein kurzes Beispiel. Betrachtet werden die zwei folgenden Regeln.

$$\begin{array}{lcl} \text{E} & \rightarrow & \text{a} \\ \text{E} & \rightarrow & \text{ab} \end{array}$$

Die Zielsymbolmenge für beide Regeln beinhalten das Nichtterminal a. Eine Grammatik, die diese beiden Regeln enthählt, kann also für ein Top-Down-Verfahren nicht verwendet werden. Grammatiken, die Linksrekursivität enthalten, können gernerell nicht verwendet werden.

Bottom-Up-Parser

Ein Bottom-Up-Parser versucht eine Rechtsableitung für den vorgelegten Satz zu konstruieren. Aus diesem Grund werden diese Parser auch *LR-Parser* genannt, wobei L für das Links-nach-Rechts-Lesen des Eingabebandes und das R für die Rechtsableitung steht.

Um die Vorgehensweise zu verstehen, folgt ein Beispiel. Betrachtet wird wieder die Grammatik in Abbildung 2.3. Diese Grammatik enthält Linksrekursionen, was bei den Grammatiken für das Top-Down-Verfahren verboten ist. Zu dem Satz $4 * 5 + 7$ kann folgende Rechtsableitung gebildet werden

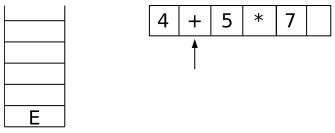
$$
\begin{aligned}
S &\rightarrow E \\
&\rightarrow E + T \\
&\rightarrow E + T * F \\
&\rightarrow E + T * number \\
&\rightarrow E + F * number \\
&\rightarrow E+ number * number \\
&\rightarrow T + number * number \\
&\rightarrow F + number * number \\
&\rightarrow number + number * number
\end{aligned}
$$

Da das Bottom-Up-Verfahren den konkreten Syntaxbaum von den Blättern zur Wurzel aufbaut, muss diese Rechtsableitung in umgekehrter Reihenfolge betrachtet werden:

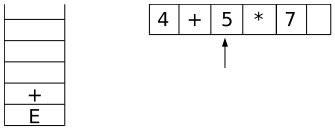
$$
\begin{aligned}
number + number * number &\rightarrow F + number * number \\
&\rightarrow T + number * number \\
&\rightarrow E + number * number \\
&\rightarrow E + F * number \\
&\rightarrow E + T * number \\
&\rightarrow E + T * F \\
&\rightarrow E + T \\
&\rightarrow S
\end{aligned}
$$

Betrachtet man den Schritt $number + number * number \rightarrow F + number * number$, fällt auf, dass die Regel $F \rightarrow number$ rückwärts angewandt wurde, man spricht in diesem Fall von einer *Reduzierung*.

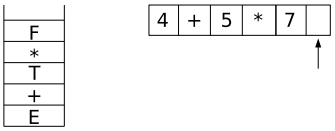
Auch das Bottom-Up-Verfahren arbeitet mit einem Kellerautomaten und einem Eingabeband, das einen Zeiger auf das eben betrachtende Symbol besitzt (kurz *aktuelles Symbol* genannt). Der Parser arbeitet für den arithemtischen Ausdruck $4+5*7$ und der oben beschriebenen Grammatik wie folgt: er liest das erste Morphem ein und legt es auf den leeren Keller ab. Der Kellerautomat besitzt mehrere Zustände, von denen die auszuführenden Aktionen abhängig sind. Nachdem er das erste Morphem gelesen hat, befindet er sich im Startzustand. Er rückt den Zeiger des Eingabebandes weiter. Das aktuelle Symbol kann noch nicht verarbeitet werden, also wird das oberste Kellersymbol zum Nichtterminal F reduziert (zur Erinnerung: die Reduzierung erfolgt mit der Regel $F \rightarrow number$). Der Zustand des Automaten ändert sich dabei. Anschließend wird das oberste Kellersymbol zu T bis schließlich E reduziert. In der folgende Abbildung ist dieser Zustand dargestellt. Auf der linken Seite ist der aktuelle Inhalt des Kellers aufgeführt und auf der rechten Seite ist das Eingabeband mit dem aktuellen Zeiger dargestellt.



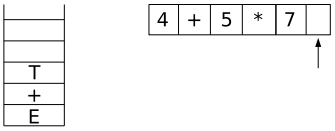
Nun befindet sich der Automat in einen Zustand, in dem er das Symbol + einlesen kann. Er setzt den Eingabezeiger weiter, legt das Symbol auf den Keller und wechselt in einen anderen Zustand.



Eine Reduzierung ist in diesem Zustand nicht möglich, daher wird ein Symbol weiter gelesen. Das setzt sich bis zur folgenden Situation fort:



In dieser Situation kann mit der Regel $T \rightarrow T * F$ reduziert werden, das heißt, die obersten drei Symbole des Kellers werden abgebaut und es wird ein T auf den Keller geschrieben. So ergibt sich folgende Situation:



Das Beispiel soll nicht zu Ende geführt werden, es soll lediglich dazu dienen, eine Vorstellung von der Vorgehensweise zu bekommen.

Es können zwei Aktionen des Parsers unterschieden werden: Reduktion und Verschiebung des Zeiger des Eingabebandes (im Englischen *Shift* genannt). Der Parser muss immer eindeutig entscheiden können, welche Aktion er ausführt. Wenn dies nicht der Fall ist, kommt es zu einem Konflikt. Es gibt zwei Arten von Konflikten: Reduktion/ Reduktion- oder Schiebe/Reduktion - Konflikt. Der erste Fall tritt bei Regeln der folgenden Art auf:

$$\begin{array}{lcl} A & \rightarrow & ab \\ B & \rightarrow & ab \end{array}$$

Wenn der Parser die Morpheme `a` und `b` auf dem Keller findet, kann er nicht entscheiden, ob er diese zum Nichtterminal `A` oder `B` reduzieren soll.

Der Schiebe/Reduktion-Konflikt tritt bei Regeln der folgenden Art auf:

$$\begin{array}{lcl} A & \rightarrow & abc \\ B & \rightarrow & ab \end{array}$$

Wenn die Morpheme `a` und `b` auf dem Keller liegen und das aktuelle Symbol auf dem Eingabeband ein `c` ist, dann kann der Parser nicht entscheiden, ob er zum Nichtterminal `B` reduzieren oder das Eingabeband um eins nach rechts schieben soll. Wenn man eine Grammatik baut oder eine vorhandene ändert, müssen diese beiden Konflikte beachtet werden. Allerdings weisen Werkzeuge, die den Parser generieren, in der Regel auf Konflikte hin, so dass eine manuelle Untersuchung ausbleiben kann.

Die Entscheidung, welche Aktion der Parser in einem bestimmten Zustand ausführt, kann durch eine sog. *Zustandübergangstabelle* festgelegt werden. Diese Tabelle ergibt sich aus einem Automaten, dessen Zustände sog. *Item-Mengen* sind. Darauf soll nicht näher eingegangen werden. In [AHO] auf den Seiten 241 bis 257 ist das Verfahren ausführlich erklärt.

2.2.3 Eigenschaften von Grammatiken

In den Abschnitten der beiden Parser-Verfahren wurde gesagt, dass nicht jede kontextfreie Grammatik für jedes Parser-Verfahren geeignet ist. Allgemein kann gesagt werden,

dass die praktikablen Verfahren nicht jede beliebige kontextfreie Grammatik verwenden können. Programmiersprachen lassen sich jedoch mit den zugelassenen Grammatiken beschreiben. Ob eine Grammatik geeignet ist, kann in der Regel nur schwer von Hand erarbeitet werden. Aus diesem Grund kann man diese Aufgabe einem Werkzeug überlassen, das entsprechende Fehlermeldungen ausgibt, wenn die Grammatik Konflikte enthält bzw. nicht geeignet ist. In diesem Fall muss die Grammatik nachgebessert werden. In Tabelle 2.1 sind die Grammatikarten den Parser-Verfahren zugeordnet.

	Top-Down-Parser	Bottom-Up-Parser
Grammatiken	LL	LR, SLR, LALR

Abkürzung	Bedeutung
LL	Eingabeband wird von **l**inks nach rechts gelesen; es wird eine **L**inksableitung erzeugt
LR	Eingabeband wird von **l**inks nach rechts gelesen; es wird eine **R**echtsableitung erzeugt
SLR	Einfaches (**s**imple) Verfahren; Eingabeband wird von **l**inks nach rechts gelesen; es wird eine **R**echtsableitung erzeugt
LALR	**L**ook**A**head; Eingabeband wird von **l**inks nach rechts gelesen; es wird eine **R**echtsableitung erzeugt

Tabelle 2.1: Grammatikarten für Parser-Verfahren

Die Mächtigkeit bzw. die Ausdrucksmöglichkeiten der Grammatiken sind in Abbildung 2.4 dargestellt.

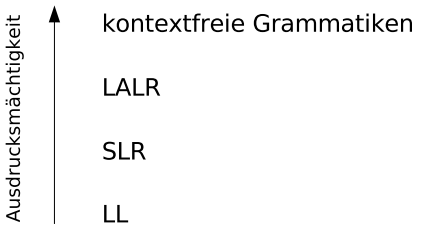


Abbildung 2.4: Ausdrucksmächtigkeit der Grammatiken

Grammatiken der LL-Sprachfamilie besitzen die geringste Ausdrucksstärke (zur Erinnerung: Linksrekursion ist nicht erlaubt). Die LALR-Grammatiken werden bevorzugt eingesetzt. Kontextfreie Grammatiken im Allgemeinen können zwar verarbeitet werden (siehe Cocke-Younger-Kasami-Algortihmus), aber dieses Vorgehen besitzt eine zu hohe Komplexität, um es praktikabel einsetzen zu können.

2.2.4 Werkzeugunterstützung

Einen Parser von Hand zu schreiben ist mitunter ein großer Aufwand, der durch die Komplexität der Grammatik drastisch erhöht werden kann.

Um das Erstellen eines Parser zu vereinfachen, wurden zahlreiche Werkzeuge geschrieben. Diese unterscheiden sich vor allem in der Grammatikunterstützung. Der Parser-Generator AntLR [Antlr] unterstützt beispielsweise nur LL-Grammatiken, was diesen Parser-Generator sehr einschränkt. Die Parser-Generatoren SableCC [GAG], ExtendedSableCC [Kon] und Beaver [Beaver] können dagegen Grammatiken von Type LALR verarbeiten. Auch unterscheiden sich die Generatoren in der Zielsprache. Alle bisher vorgestellten erzeugen einen Parser in der Programmiersprache Java. AntLR unterstützt allerdings noch andere Sprachen. Manche von den Generatoren unterstützen auch die Definition der Morpheme (wie AntLR, SableCC und ExtendedSableCC). Eine weitere nützliche Funktion ist das Erzeugen eines abstrakten Syntaxbaumes (siehe 2.4) und das Einbringen von Quelltext in die Spezifikation. Tabelle 2.2 fasst die Eigenschaften zusammen.

	SableCC	ExtSableCC	Beaver	AntLR
Grammatik-Typ	LALR	LALR	LALR	LL
Sprache des generierten Parsers	Java	Java	Java	Java und andere
Morphem-Spezifikation	ja	ja	nein	ja
Quelltext	nicht direkt in der Spezifikation	nicht direkt in der Spezifikation	ja	ja
Erzeugung eines abstrakten Syntaxbaumes	ja	durch Framework	ja	ja

Tabelle 2.2: Vergleich einiger Parser-Generatoren

2.3 Attributierung

Die syntaktische Analyse erzeugt aus dem Quelltext einen konkreten Syntaxbaum. Oft möchte man diesen transformieren, zum Beispiel nach Assemblercode oder in einen abstrakten Syntaxbaum (siehe Abschnitt 2.4). Während der Transformation können Berechnungen einfließen, die sich über die Struktur des konkreten Syntaxbaumes ergeben.

Transformationen dieser Art können mittels der sog. *Attributierung* durchgeführt werden. Dieser Abschnitt teilt sich in zwei kleinere Teile. Zum einen werden die Grundlagen für die Attributierung gelegt, die mit einem kleinen Beispiel veranschaulicht werden, und zum anderen werden ein paar Worte zur praktischen Umsetzung bezüglich dieser Arbeit erläutert.

2.3.1 Grundlagen

Für eine Baumtransformation sind drei Dinge notwendig:

- eine kontextfreie Grammatik

- Attribute
- Attributauswertungsregeln

Alle Grammatiksymbole (Nichtterminale und Terminale) bekommen Attribute zugewiesen, die einen bestimmten Typ haben. Eine Grammatik kann zum Beispiel das Nichtterminal `Z` mit dem Attribut `length` besitzen, was eine natürliche Zahl repräsentiert. Eine Attributauswertungsregel bezieht sich immer auf eine Regel der Grammatik. Innerhalb der Attributauswertungsregel können die Attribute der Grammatiksymbole verwendet werden. Es können Aktionen in die Regeln hineingeschrieben und Attributen können Werte zugewiesen werden (was nicht mit allen Attributen möglich ist, siehe unten).

Beispiel Als Beispiel soll eine angepasste Grammatik aus [Schmitz] Seite 96 dienen. Die folgende Grammatik beschreibt Tenärzahlen und ihre Attributierung transformatiert eine Tenär- in eine Dezimalzahl, außerdem wird die Länge der Tenärzahl zurückgegeben. In Abbildung 2.5 werden alle Bestandteile der Grammatik definiert. Unter jeder Regel befinden sich die Attributauswertungsregln.

In der Grammatikregel $Order[1] \rightarrow Order[2]Digit$ wurden den Nichtterminalen `Order` Zahlen zugeordnet, um auf sie in den Attributierungsregeln zugreifen zu können. Sie sind nicht Teil der Grammatikspezifikation, sondern dienen ledliglich der Unterscheidung.

Die Auswertung der Attribute beginnt von der Wurzel aus und führt bis zu den Blättern, was nicht allgemein der Fall ist. Die Traversierung geschieht in einer Pre-Order-Reihenfolge wie bei einer Tiefensuche. So wird der Baum von links nach rechts durchlaufen. Nicht alle Attribute können an jeder Stelle sofort ausgewertet werden, da sie von anderen Attributen, die noch berechnet werden müssen, abhängen. Um dies zu verdeutlichen wird die zweite Grammatikregel mit ihrer Attributierung betrachtet:

Order[1]	→	Order[2] Digit
Order[2].weight	:=	Order[1].weight *3
Order[1].value	:=	Order[2].value + Digit.value
Order[1].length	:=	Order[2].length + Digit.length

Die Attribute `length` und `value` des Nichtterminals `Order[1]` hängen von den Attributen der Nichtterminale `Order[2]` und `Digit` ab, so dass sie noch nicht berechnet werden können, wenn sich die Auswertung am Anfang befindet. Das Attribut `weight` von `Order[2]` kann allerdings belegt werden. Dieses Attribut ist ein sog. *ererbtes Attribut*, es wird von "oben nach unten" durch den Baum gereicht. Terminale können keine ererbte Attribute besitzen. Die Attribute `length` und `value` dagegen sind sog. *synthetisierte Attribute*. Sie werden von "unten nach oben" durch den Baum gereicht.

In diesem Beispiel wird das ererbte Attribut `weight` auf dem Weg zu den Blättern berechnet. An den Blättern angekommen, wird es genutzt, um der einzelnen Ziffer eine Wertigkeit zuzuordnen. Diese wird als synthetisiertes Attribut zurückgegeben. Die Wertigkeiten werden nach und nach aufaddiert, bis der endgültige Dezimalwert an der Wurzel feststeht. Mit der Länge der Zahl verhält es sich ein klein wenig anders. Hier wird kein ererbtes Attribut benötigt, statt dessen "kennt" jede Ziffer ihre Länge, die als synthetisiertes Attribut zurückgegeben wird. Die Längen werden nach und nach aufaddiert, bis sie oben an der Wurzel ankommen, wo sie die gesamte Länge der tenären Zahl bilden.

Die Traversierung des Baumes kann auch in einer anderen Reihenfolge geschehen. Dann muss auf *zyklische Abhänigkeiten* der Attribute geachtet werden, die mitunter

G = $(N, \Sigma, P, Number)$
N = $\{Number, Order, Digit\}$
Σ = $\{0, 1, 2\}$

P:

Number	→	Order
Order.weight	:=	1
Number.value	:=	Order.value
Number.length	:=	Order.length

Order[1]	→	Order[2] Digit
Digit.weight	:=	Order[1].weught
Order[2].weight	:=	Order[1].weight * 3
Order[1].value	:=	Order[2].value + Digit.value
Order[1].length	:=	Order[2].length + Digit.length

Order	→	Digit
Digit.weight	:=	Order.weight
Order.value	:=	Digit.value
Order.length	:=	1

Digit	→	0
Digit.value	:=	0 * Digit.weight

Digit	→	1
Digit.value	:=	1 * Digit.weight

Digit	→	2
Digit.value	:=	2 * Digit.weight

Abbildung 2.5: Beispiel-Attributierung

nicht aufgelöst werden können. In Abbildung 2.6 ist dazu ein einfaches Beispiel dargestellt.

A	→	B C D
B.x	:=	C.y
C.y	:=	B.x

Abbildung 2.6: Zyklische abhängige Attribute

Die Nichtterminale `B` und `C` besitzen die Attribute `x` und `y`. Beide Attribute sind ererbt. Das Attribut `x` hängt von `y` ab, kann also erst berechnet werden, wenn `y` berechnet wurde. Aber `y` hängt auch von `x` ab. Es liegt eine zyklische Abhängigkeit vor, die in diesem Fall nicht aufgelöst werden kann.

2.3.2 Praktische Umsetzung

In dieser Arbeit wird die Attributierung genutzt, um einen konkreten Syntaxbaum in einen abstrakten Syntaxbaum und um einen abstrakten Syntaxbaum in einen abstrakten Syntaxgraphen zu transformieren. Die Traversierung des Baumes erfolgt mit rekursiven Methodenaufrufen, wobei die Parameter der Methode den ererbten und die Rückgabewerte den synthetisierten Attributen entsprechen.

Zu zyklischen Abhängigkeiten kommt es dabei nicht, da folgende Annahme getroffen wird: auf deklarierte Elemente der Sprache kann nur dann zugegriffen werden, wenn diese Elemente **vorher** deklariert wurden. Das heißt Konstrukte wie

$$\text{x = 1; int x;}$$

(angelehnt an die Programmiersprache Java) sind nicht erlaubt. Solche Konstrukte bedingen, dass Attribute berechnet werden müssen, ohne die vorhergehenden Attribute der Regel zu berechnen. Für das Beispiel könnte es folgende Grammatikregel geben

$$\textit{Statements} \rightarrow \textit{Assign Declaration}$$

In diesem Fall müssten die Attribute von `Declaration` vor dem Auswerten der Attribute von `Assign` berechnet werden, um zu überprüfen, ob es sich bei `x` um eine deklarierte Variable handelt.

2.4 Die abstrakte Syntax

In diesem Abschnitt soll veranschaulicht werden, was unter der *abstrakten Syntax* zu verstehen ist. Im Kapitel über die Sprache PL0 (siehe Kapitel 3 auf Seite 43) wird eine abstrakte Syntax eingeführt. Auch benötigt der OCL-Parser (siehe Kapitel 6 auf Seite 99) eine abstrakte Syntax. Durch den Einsatz einer abstrakten Syntax wird die Verarbeitung eines konkreten Programms, eines konkreten Ausdruckes, ... wesentlich vereinfacht, da eine Abstraktion von der konkrete Syntax vorgenommen wird.

Dieses Kapitel ist wie folgt gegliedert: zunächst wird der Begriff der abstrakten Syntax eingeführt, um anschließend auf Referenzen in einem abstrakten Syntaxbaum einzugehen. Da der Begriff des abstrakten Syntaxgraphen im weiteren Verlauf dieser Arbeit eine große Bedeutung besitzt, wird dieser in einem eigenen Abschnitt vorgestellt. Zuletzt wird auf eine andere Darstellungsform der abstrakten Syntax hingewiesen und es wird der Begriff der semantischen Regeln eingeführt. Um Verwirrung des Begriffes *Metamodell* zu vermeiden, wird versucht in einem Abschnitt den Begriff zu anderen Gebrauchsformen abzugrenzen.

2.4.1 Der Begriff der abstrakten Syntax

Die konkrete Syntax einer Sprache wird mit Hilfe von kontextfreien Grammatiken beschrieben. Ein Parser-Generator baut daraus ein Programm, was einen Quelltext, der in der beschriebenen Sprache geschrieben ist, einliest und ihn in einen konkreten Syntaxbaum transformiert. Der konkrete Syntaxbaum enthält überflüssige Elemente, die sich zum Beispiel durch die Definition mathematischer Ausdrücke, die in der Grammatik definiert wurden, ergeben. Dazu soll folgende kleine Grammatik betrachtet werden (entnommen aus [AHO]), die in Abbilding 2.7 dargestellt ist.

Mit Hilfe dieser Grammatik können arithmetische Ausdrücke gebildet werden, die die Operatoren + und * enthalten. Die Operatoren + und * sind nicht in der gleichen Regel

E	→	E + T
E	→	T
T	→	T * F
T	→	F
F	→	number

Abbildung 2.7: Grammatik mathematischer Ausdrücke

definiert, sondern auf verschiedenen Ebenen. Um einen Ausdruck mit einem * zu erzeugen, muss erst der Umweg über die Regel `E → T` gegangen werden. Diese Gliederung, auch *semantisches Niveau* genannt, dient dazu, den Operatoren Prioritäten zuzuordnen. Das höchste semantische Niveau, in diesem Fall +, besitzt die niedrigste Priorität. Wenn ein mathematischer Ausdruck in einen konkreten Syntaxbaum transformiert wird, ergeben sich mitunter sehr lange Ketten von Knoten. Ein Knoten innerhalb einer Kette hat genau einen Vorgänger- und Nachfolgerknoten. Dies soll an einem Beispiel verdeutlicht werden. Betrachtet wird der folgende mathematische Ausdruck:

$$5 + 6 * 7 + 4$$

Aus diesem Ausdruck wird der in Abbildung 2.8 dargestellte konkrete Syntaxbaum. Die Ketten in diesem Baum sind fett markiert.

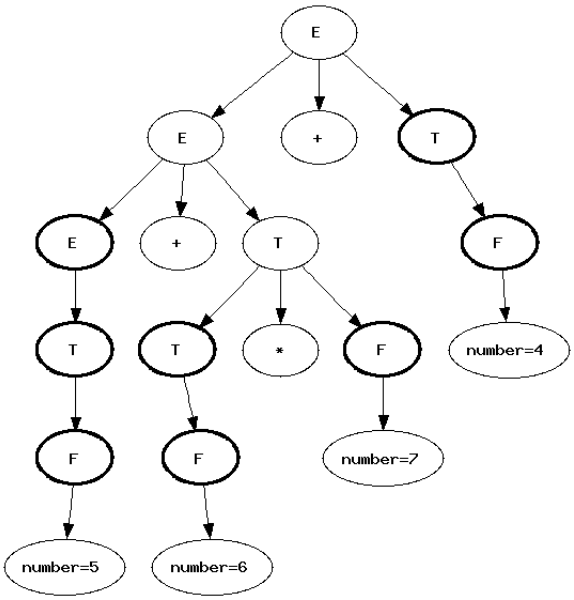


Abbildung 2.8: Konkreter Syntaxbaum des mathematischen Ausdruckes $5 + 6 * 7 + 4$

Der konkrete Syntaxbaum ist sehr unhandlich und benötigt viel Speicher. Zu dem lassen sich die Ketten entfernen, so dass sich der in Abbildung 2.9 dargestellte Baum ergibt.

Wie zu erkennen ist, rutschen die Operatoren eine Stufe nach oben und die Ketten verschwinden. Dieser Baum wird *abstrakter Syntaxbaum* genannt (in diesem Fall kann

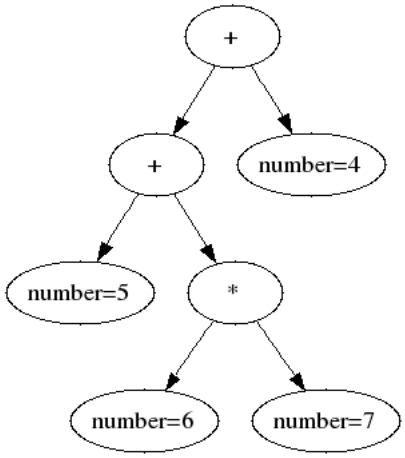


Abbildung 2.9: Abstrakter Syntaxbaum des mathematischen Ausdruckes $5 + 6 * 7 + 4$

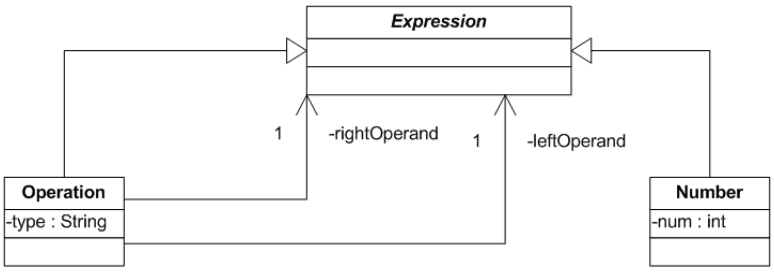


Abbildung 2.10: UML-Klassendiagramm für mathematische Ausdrücke

auch *Formelbau* gesagt werden, da es sich hierbei um einen mathematischen Ausdruck handelt).

Zusammenfassend lässt sich sagen: der abstrakte Syntaxbaum entsteht aus dem konkreten Syntaxbaum, indem Ketten beseitigt werden.

Diesen abstrakten Syntaxbaum kann man auch anders auffassen. Die Operatoren + und * stellen nichts anderes als mathematische Operationen dar, die zwei Parameter besitzen. Als Parameter kann nun eine konkrete Zahl oder ein anderer mathematischer Ausdruck in Frage kommen. Es kann auch gesagt werden, dass eine konkrete Zahl ein mathematischer Ausdruck ist. Solche Kategoriekonzepte lassen sich sehr schön durch UML-Diagramme darstellen. Um mathematische Ausdrücke zu modellieren, benötigt man die Klassen `Operation`, `Number` und `Expression`. Eine Operation besitzt zwei Parameter vom Typ `Expression`. Eine `Expression` kann entweder eine Zahl sein oder wieder eine Operation. In der Abbildung 2.9 besitzt das erste Plus eine weitere Operation Plus. Eine mögliche Klassenhierarchie könnte damit wie in Abbildung 2.10 aussehen.

Es stellt sich heraus, dass sich hier das Baummuster (im Englischen *Composite-Pattern* genannt) aus [GOF] versteckt. Zwei Informationen fehlen noch: der Typ der Operation und der Wert der Zahl. Beide werden mittels Attributen modelliert. Die Klasse `Operation` bekommt ein Attribut `type`, das den Typen repräsentiert (+ oder *). Die

$$
\begin{array}{lcl}
\mathrm{E} & \rightarrow & \mathrm{E} + \mathrm{T} \\
\mathrm{E} & \rightarrow & \mathrm{T} \\
\mathrm{T} & \rightarrow & \mathrm{T} * \mathrm{F} \\
\mathrm{T} & \rightarrow & \mathrm{F} \\
\mathrm{F} & \rightarrow & \text{number} \\
\mathrm{F} & \rightarrow & \text{id}
\end{array}
$$

Abbildung 2.11: Grammatik mathematischer Ausdrücke mit Variablen

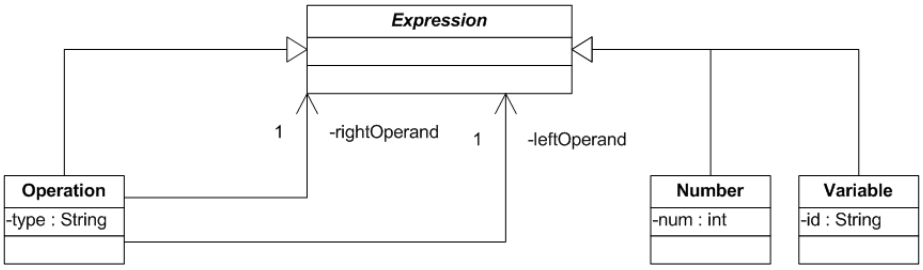


Abbildung 2.12: UML-Klassendiagramm für mathematische Ausdrücke mit Variablen

Klasse `Number` speichert den konkreten Zahlenwert im Attribut `num`.

Was ist durch diese Darstellung passiert? Von dem abstrakten Syntaxbaum eines beliebigen mathematischen Ausdruckes wurde eine kleine Klassenhierarchie entwickelt. Diese repräsentiert nicht einen mathematischen Ausdruck, sondern **alle**, die mit der kleinen Beispielgrammatik (siehe Abbildung 2.7) erzeugt werden können. Man kann auch sagen, dass der abstrakte Syntaxbaum eine Instanz der Klassenhierarchie ist. Das führt hin zu der Idee der *Metamodelle*. Die Klassenhierarchie ist ein Metamodell aller mathematischen Ausdrücke.

Die mathematischen Ausdrücke werden dabei durch abstrakte Konzepte beschrieben, in diesem Fall durch `Expression`, `Operation` und `Number`. Man kann diesem Metamodell einen Sinn geben, während ein konkreter Syntaxbaum nur ein Baum ist, an dem viele Knoten hängen.

2.4.2 Referenzen im abstrakten Syntaxbaum

In Programmiersprachen gibt es in mathematischen Ausdrücken neben den Zahlen auch Variablen. Diese Variablen werden in einem Deklarationsteil im Programm deklariert und sie tauchen dann an beliebiger Stelle im Programm auf.

Um auch Variablen mit Hilfe der vorhergehenden Grammatik darstellen zu können, muss sie ein wenig erweitert werden (siehe Abbildung 2.11).

Das Morphem `id` steht dabei für einen Identifizierer, der mit einem Buchstaben beginnen muss und beliebig viele Zahlen und Buchstaben enthalten darf. Das Metamodell der mathematischen Ausdrücke ändert sich ebenfalls. Eine neue Klasse `Variable` wird hinzugefügt, die ein Attribut mit dem Namen des Identifizierers aufnimmt. Das Modell ist in Abbildung 2.12 dargestellt.

Die Variable kann also auch ein Blatt des Baumes werden.

Aber ist es noch ein Baum? Für alle Ausdrücke der Beispielgrammatik trifft dies

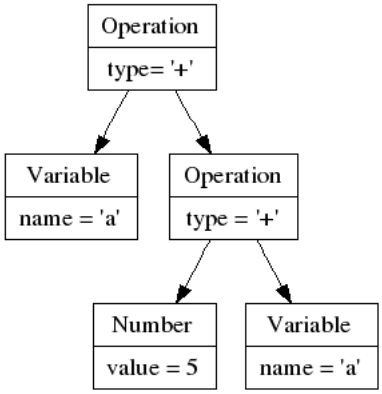


Abbildung 2.13: Abstrakter Syntaxbaum mit Variable

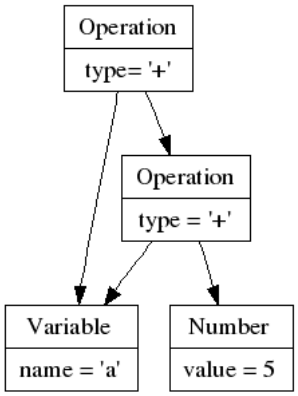


Abbildung 2.14: Abstrakter Syntaxbaum mit nur einer Variableninstanz

zu (siehe Abbildung 2.7). Es gibt aber eine Beobachtung, die gemacht werden kann. Betrachtet werde der folgende mathematische Ausdruck:

$$a + 5 + a$$

Daraus ergibt sich der in Abbildung 2.13 dargestellte abstrakte Syntaxbaum.

Die Darstellung des abstrakten Syntaxbaumes wurde geändert. Statt nur die Operationen und die Werte der Zahlen einzuzeichnen, sollen hier die Instanzen der Klassen betrachtet werden. Es fällt auf, dass es zwei Instanzen der Klasse `Variable` gibt, die denselben Namen tragen. In dem mathematischen Ausdruck bezeichnet `a` aber genau eine Variable. So kann der abstrakte Syntaxbaum zu der in Abbildung 2.14 dargestellten Struktur geändert werden.

Was ist hier passiert? Statt jedem Identifizierer ein eigenes Objekt zuzuordnen, wird für einen Identifizierer genau ein Objekt erzeugt. Alle weiteren Vorkommnisse dieses Identifizierers werden auf dieses Objekt verwiesen.

In diesem Moment wurde der abstrakte Syntaxbaum mit Semantik angereichert. In dem mathematischen Ausdruck $a + 5 + a$ steht sowohl das linke `a` als auch das rechte

für denselben Wert und genau dieses drückt sich nun im abstrakten Syntaxbaum aus.

2.4.3 Abstrakter Syntaxgraph

Programmiersprachen sind in der Regel komplizierter aufgebaut. Daher kann es passieren, dass aus einem konkreten Quelltext kein abstrakter Syntaxbaum sondern ein abstrakter Syntax**graph** wird. Dazu ein kleines Beispiel: Viele Programmiersprachen stellen Prozeduren bereit, die mehrere Anweisungen zusammenfassen. Eine der Anweisungen kann eine Prozedur aufrufen. Zusammengefasst ist das kleine Szenario im Klassendiagramm der Abbildung 2.15.

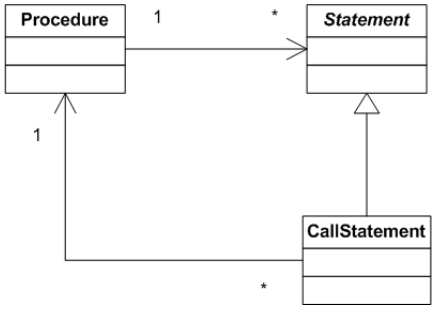


Abbildung 2.15: UML-Diagramm von Prozeduren

Die Klasse `CallStatement` besitzt eine Referenz auf die verweisende Prozedur. In einem rekursiven Aufruf bildet sich ein Zyklus. Die Situation ist in Abbildung 2.16 dargestellt.

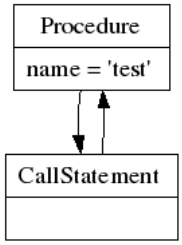


Abbildung 2.16: rekursiver Prozeduraufruf

Da nun ein Zyklus in dem Objektgraph vorkommt, handelt es sich hierbei nicht mehr um einen abstrakten Syntaxbaum, sondern um einen abstrakten Syntaxgraph.

2.4.4 Darstellung der abstrakten Syntax mittels einer kontextfreien Grammatik

Wie gezeigt, kann die abstrakte Syntax durch UML-Diagramme beschrieben werden. Auch wurde gezeigt, dass Instanzen der Klassenhierarchien abstrakte Syntaxbäume oder abstrakte Syntaxgraphen sein können.

Expression	→	Operation | Number | Variable
Operation	→	Expression Type Expression
Number	→	num
Variable	→	id
Type	→	type_name

Abbildung 2.17: Abstrakte kontextfreie Grammatik der mathematischen Ausdrücke

Die abstrakte Syntax kann auch durch kontextfreie Grammatiken beschrieben werden (wie es beispielsweise in Abschnitt 6.2.2 auf Seite 104 umgesetzt wird). Für die abstrakte Syntax in Abbildung 2.12 ergibt sich die in Abbildung 2.17 dargestellte kontextfreie Grammatik.

Instanzen dieser Darstellung sind ausschließlich abstrakte Syntaxbäume. Ein Graph lässt sich mit dieser Darstellung nicht erzeugen.

Zu Beginn dieses Kapitels wurde gesagt, dass ein abstrakter Syntaxbaum teilweise Bedeutung trägt und nicht nur ein Konstrukt aus Knoten und Kanten wie ein konkreter Syntaxbaum. Diese Aussage lässt sich auch auf die abstrakte kontextfreie Syntax beziehen. Betrachtet man die erste Regel der Beispielgrammatik

Expression → Operation | Number | Variable

können die Nichtterminale `Operation`, `Number` und `Variable` als Ausdrücke (`Expression`) aufgefasst werden, so wie die Vererbungsbeziehung in Abbildung 2.12. Die zweite Regel

Operation → Expression Type Expression

sagt aus, dass sich das Konstrukt `Operation` aus drei anderen Konstrukten zusammensetzt, nämlich aus zwei Ausdrücken (`Expression`) und einer Typangabe (`Type`). Im Klassendiagramm wird dies durch Assoziationen ausgedrückt. Die Morpheme bzw. Terminale werden zu Attributen (`id`, `num` und `type_name`). Somit lässt sich eine Vererbungs- und Kompositionsstruktur innerhalb der abstrakten kontextfreien Grammatik erkennen.

Die Beispielgrammatik in Abbildung 2.17 erzeugt ähnlich komplexe Bäume wie die konkrete Grammatik in Abbildung 2.11. In diesem Fall ist das Beispiel der mathematischen Ausdrücke schlecht gewählt. Wenn man an andere Sprachen denkt, die Schlüsselwörter und Begrenzer besitzen, kann ein Baum der abstrakten Syntax weniger komplexer werden, da die Schlüsselwörter und Begrenzer in der abstrakten Syntax entfallen.

2.4.5 Semantische Regeln

Wenn ein konkreter Syntaxbaum in einen abstrakten transformiert wurde, möchte man diesen oft in einen abstrakten Syntaxgraphen übersetzen. Während dieser Übersetzung sollen meist semantische Regeln (im Englischen *wellformdness rules* genannt) umgesetzt werden. In Abbildung 2.13 ist ein abstrakter Syntaxbaum dargestellt, der zwei Variableobjekte besitzt, die aber eigentlich dasselbe Objekt bezeichnen sollen. Um diesen Baum in einen Graphen zu transformieren, bedient man sich dem Interpretermuster aus [GOF], das heißt, es wird eine Umgebung (im Englischen *Environment*) eingeführt, die während der rekursiven Auswertung genutzt wird, um die bisherigen Variablen aufzulesen. Wenn die Rekursion auf eine Variable stößt, wird geprüft, ob diese Variable

in die Umgebung schon eingetragen wurde. Wenn ja, dann wird sie durch das Objekt in der Umgebung ersetzt, andernfalls wird es in die Umgebung neu aufgenommen. Mit diesem Ansatz können noch andere semantische Regeln geprüft werden. Beispielsweise kann neben dieser *Namensauflösung* auch eine Typprüfung stattfinden.

2.4.6 Der Begriff Metamodell

Der Begriff Metamodell wurde in diesem Kapitel etwas unüblich verwendet. Im Abschnitt 1.1 auf Seite 13 wird der Begriff Metamodell in einen anderen Zusammenhang gesetzt. Dort stellen Klassendiagramm *Modelle* dar, während sie in diesem Kapitel als *Metamodelle* bezeichnet werden. Die Ausführungen in diesem Kapitel lehnen sich an [GSCK] auf den Seiten 285 bis 298 an. Dort wird im Zusammenhang mit der abstrakten Syntax der Begriff Metamodell für die Klassendiagramme verwendet.

Zunächst sollte festgehalten werden, dass mit einem Metamodell immer der Umfang einer Sprache auf abstrakter Ebene beschrieben wird. Die abstrakte Syntax stellt somit ein Metamodell dar, da sie eine Sprache beschreibt (in den Beispielen die Sprache aller mathematischen Ausdrücke). Die abstrakte Syntax kann aber mit Hilfe von Klassendiagrammen dargestellt werden. Dabei nehmen diese nur eine beschreibende Position ein. Ein Metamodell kann neben der abstrakten Beschreibung einer Sprache auch semantische Regeln beinhalten. Die semantischen Regeln können dabei mittels OCL beschrieben werden.

Ein UML-Klassendiagramm beschreibt für einen Teil einer Software dagegen einen Ausschnitt aus der Realität. Beispielsweise kann mit einem UML-Diagramm die Domäne einer Universität beschrieben werden, mit den Entitäten Studenten, Fakultät, Professoren etc. Das heißt, dieser Ausschnitt abstrahiert einen Teil der Realität und stellt somit ein Modell dar. In diesem Fall wird ein Modell mittels der UML dargestellt.

2.4.7 Zusammenfassung

Ein Parser erzeugt mit Hilfe einer konkreten kontextfreien Grammatik aus einem Quelltext einen konkreten Syntaxbaum. Dieser konkrete Syntaxbaum enthält oft nutzlose Elemente und beschreibt sehr umständlich ein Programm, einen Ausdruck, ... Daher transformiert man diesen in einen abstrakten Syntaxbaum, dessen Knoten durch Instanzen von Klassen dargestellt werden können. Die zugehörigen Klassenbeziehungen, dargestellt beispielsweise in einem UML-Diagramm, abstrahieren von der konkreten Syntax in der Art, dass nur noch die wesentlichen Konzepte einer Sprache erfasst werden. Oft möchte man aus einem abstrakten Syntaxbaum einen abstrakten Syntaxgraphen machen, da dieser Objekte mittels Referenzen verbinden kann. Diese Verbindungen drücken Semantik aus. Während solch einer Transformation werden semantische Regeln geprüft.

Die abstrakte Syntax kann durch Klassendiagramme dargestellt werden (man nennt diese dann *Metamodell*) oder durch kontextfreie Grammatiken. Letztere lassen sich in ein UML-Diagramm überführen.

3 PL0

PL0 wurde von Niklaus Wirth entwickelt [Wirth], um eine einfache Sprache für das Erlernen des Parserbaus zur Verfügung zu stellen. PL0 ist an die Programmiersprache PASCAL angelehnt, besitzt im Gegensatz dazu keine Parameterübergabe, keine Typen, Zeiger und keine Funktionen. PL0 wurde in dieser Arbeit als Beispielimplementierung gewählt, um einen Überblick über die Parser-Generatoren SableCC, die Erweiterung von SableCC, Beaver und dem Attributauswerter JastAdd zu bekommen. Auch dient die Sprache als Grundlage für die Beispiele, die im SableCC- und JastAdd-Kapitel verwendet werden.

Aus diesem Grund wird PL0 in diesem Kapitel näher vorgestellt. Zunächst wird die Sprache PL0 informell beschrieben. Anschließend werden die verwendeten Morphemklassen aufgezeigt. Daran schließt sich die Beschreibung der konkreten Syntax an. Ziel der Arbeit ist es den OCL2-Parser des Dresdener OCL-Toolkits an die Pivotschnittstelle [Braeuer] anzupassen. Dafür ist es notwendig den Begriff der abstrakten Syntax zu verstehen (siehe Abschnitt 2.4 auf Seite 34). Für die Beispielsprache PL0 wird eine abstrakte Syntax vorgestellt. Das Kapitel endet mit einer Attributierungsbeschreibung für die Sprache, wobei diese zum Ziel hat, ein abstraktes Modell für ein gegebenes Programm zu erzeugen.

3.1 Vorstellung der Sprache PL0

In diesem Abschnitt wird die Sprache PL0 vorgestellt. Um einen ersten Eindruck von der Sprache zu bekommen, wird ein kleines einführendes Beispielprogramm besprochen. Anschließend wird das *Blockniveau* vorgestellt. Daran schließen sich die einzelnen Elemente von PL0 an: Konstanten- und Variablendeklaration, die Prozedurdeklaration und die Anweisungen.

3.1.1 Einführung anhand eines Beispielprogramms

Um PL0 vorzustellen, wird ein kleines Beispielprogramm betrachtet, das die Fakultät berechnet (siehe Listing 3.1).

```
1  VAR n, r;
2  PROCEDURE fac;
3          BEGIN
4          IF n#0 THEN
5                  BEGIN
6                  r := r * n;
7                  n := n - 1;
8                  CALL fac;
9                  END;
10         END;
11
12 BEGIN
```

```
13 r := 1;
14 ? n;
15 CALL fac;
16 ! r;
17 END.
```

Listing 3.1: PL0-Programm zur Berechnung der Fakultät

Das Beispielprogramm gliedert sich in drei Teile: der Deklaration von globalen Variablen (Zeile 1), der Prozedur `fac` (Zeile 2-10) und dem Hauptprogramm (Zeile 12-17). Die globalen Variablen `n` und `r` besitzen als Typ die ganzen Zahlen, wie alle Variablen und Konstanten (im Beispiel nicht gezeigt). Globale Variablen und Konstanten sind innerhalb des gesamten Programms sichtbar, es sei denn, sie werden verdeckt (siehe weiter unten). Die Prozedur `fac` besteht hier nur aus dem Anweisungsteil (Zeilen 3-10). Sie prüft, ob `n` einen Wert ungleich 0 besitzt. Wenn ja, wird `r` mit dem aktuellen Wert von `n` multipliziert, `n` wird um eins subtrahiert und die Prozedur ruft sich selbst wieder auf. Der rekursive Aufruf dauert so lange an, bis `n` auf 0 gesunken ist.

In Zeile 12 beginnt das Hauptprogramm: es setzt die Variable `r` auf den Wert 1 (Zeile 13), verlangt nach einer Benutzereingabe (Zeile 14), die in der Variablen `n` gespeichert wird, die Prozedur `fac` wird aufgerufen (Zeile 15) und letztendlich wird der berechnete Wert ausgegeben (Zeile 16).

Zu beachten ist, das jede Anweisung mit einem Semikolon ; abgeschlossen wird.

3.1.2 Blockniveau

In einem PL0-Programm finden sich sog. *Blockniveaus*. Ein Blockniveau beschreibt dabei den Gültigkeitsbereich von Variablen, Konstanten und Prozeduren. Diese Gültigkeitsbereiche können ineinander geschachtelt werden, so dass sich Variablen-, Konstanten- und Prozedurdeklarationen verdecken können (siehe unten).

An zwei Stellen kann ein neues Blockniveau eingefügt werden. Zum einen besitzt jedes PL0-Programm ein globales Blockniveau. In diesem befindet sich das Hauptprogramm. Zum anderen führt jede Prozedurdeklaration ein neues Blockniveau ein.

Beispiele folgen in den nächsten Abschnitten.

3.1.3 Konstanten- und Variablendeklarationen

Konstantendeklaration

Eine Konstantendeklaration wird durch das Schlüsselwort *CONST* eingeleitet. Anschließend folgen die Konstantennamen und ihre Werte, jeweils getrennt durch ein Komma. Abgeschlossen wird die Konstantendeklaration mit einem Semikolon ;. In Listing 3.2 ist ein Beispiel dargestellt, in dem die Konstanten `a`, `b` und `c` mit den Werten 5, 1 und 10 deklariert werden.

```
CONST a = 5, b = 1, c = 10;
```

Listing 3.2: Beispiel für eine Konstantendeklaration

Konstanten besitzen als Wertebereich die ganzen Zahlen, andere Werte sind nicht zugelassen. Die Konstantendeklaration erfolgt vor der Variablendeklaration und kann nach der Definition einer Prozedur stehen. Globale Konstanten können deklariert werden, indem sie im äußersten Block untergebracht werden.

Die Konstantennamen müssen sich von Prozedurnamen unterscheiden, die vorher deklariert wurden. Konstanten können Konstanten und/oder Variablen überdecken, wenn die betreffende Konstante oder Variable ein höheres Blockniveau besitzt. Dazu zwei Beispiele: in Listing 3.3 wird eine globale Variable `a` deklariert und von der Konstanten `a` überdeckt, weil die Konstante in einem tieferen Blockniveau liegt. In Listing 3.4 wird die Prozedur `A` definiert. Die Konstante `A` erhält denselben Namen wie die zuvor definierte Prozedur. Aus diesem Grund ist der Name *A* für die Konstante nicht zulässig.

```
VAR a;
PROCEDURE X;
        CONST a = 5;
...
```

Listing 3.3: Konstantendeklaration mit erlaubter Überdeckung

```
PROCEDURE A
CONST A = 5;
...
```

Listing 3.4: Konstantendeklaration ohne erlaubter Überdeckung

Variablendeklaration

Eine Variabledeklaration wird mit dem Schlüsselwort *VAR* eingeleitet. Darauf folgen die Variablennamen durch Kommata getrennt. Die Deklaration wird mit einem Semikolon ; abgeschlossen. Ein Beispiel ist in Listing 3.5 dargestellt. Es werden die Variable `x`, `y` und `z` deklariert.

```
VAR x, y, z;
```

Listing 3.5: Beispiel für eine Variablendeklaration

Zu beachten ist, das sich die Variablennamen von vorhergehenden Prozedurnamen unterscheiden müssen. Da die Variablen nach den Konstanten deklariert werden, darf eine Variable nicht denselben Namen wie eine Konstante im selben Blockniveau erhalten. Wenn eine Konstante mit demselben Namen in einem höheren Blockniveau deklariert wurde, wird sie von der neuen Variablen überdeckt. Dies trifft auch auf Variablen zu, die sich ebenfalls in einem höheren Blockniveau befinden. Dazu zwei Beispiele: in Listing 3.6 ist eine erlaubte Variablendeklaration dargestellt, die die gleichnamige Konstante in einem höherem Blockniveau überdeckt; in Listing 3.7 dagegen wird eine Variablendeklaration dargestellt, die nicht erlaubt ist.

```
CONST a = 5;
PROCEDURE X;
        VAR a;
...
```

Listing 3.6: Variablendeklaration mit erlaubten Überdeckung

```
CONST a = 5;
VAR a;
...
```

Listing 3.7: Variablendeklaration ohne erlaubten Überdeckung

3.1.4 Prozedurdeklaration

Prozeduren dienen dazu eine oder mehrere Anweisungen aufzunehmen, um sie an beliebiger Stelle mittels des Befehls `call` aufrufen zu können. Eine Prozedur besitzt einen Namen, sie kann Konstanten und/oder Variablen enthalten und muss mindestens eine Anweisung beinhalten. Listing 3.8 zeigt eine Prozedur `double`, deren Aufgabe es ist, einen Wert in Form der globalen Variable `input` zu duplizieren.

```
1 PROCEDURE double;
2 CONST d = 2;
3 VAR result;
4         result = input * d;
```

Listing 3.8: Prozedur `double`

Zu beachten ist, dass die Zuweisung mit einem Semikolon abgeschlossen wird. Wenn eine Prozedur mehrere Anweisungen beinhalten soll, dann müssen diese durch die Schlüsselwörter `BEGIN` und `END` eingeschlossen werden (siehe einleitendes Beispiel Listing 3.1). Auch hier gilt, das auf das Schlüsselwort `END` ein Semikolon folgt.

Prozeduren können ineinander geschachtelt werden. Die inneren Prozeduren folgen auf die Variablendeklaration. In Listing 3.9 ist dazu ein Beispiel dargestellt.

```
1  PROCEDURE A;
2  CONST a = 5;
3  VAR b;
4
5          PROCEDURE A1;
6          b = 1;
7
8          PROCEDURE A2;
9          VAR b, c;
10
11                 PROCEDURE A2A;
12                 b = 5;
13
14         c = 1;
15
16 BEGIN
17 ?b;
18 CALL A1;
19 END;
```

Listing 3.9: verschachtelte Prozeduren

Die Prozedur `A` beinhaltet die Prozeduren `A1` und `A2`. Die Prozedur `A2` besitzt wiederum die Prozedur `A2A`. Jede Prozedur kann ihre eigenen Konstanten oder Variablen deklarieren, wobei es zu Überdeckungen (siehe Abschnitt 3.1.3 auf Seite 45) kommen kann.

Jede Prozedur führt ein neues Blockniveau ein. Das Blockniveau beschreibt den Gültigkeitsbereich von Konstanten, Variablen und Prozeduren. Eine Variable oder Konstante, die durch eine Prozedur deklariert wird, ist auch in niedrigeren Blockniveaus gültig, sofern sie nicht überdeckt wird. Auf die deklarierten Prozeduren kann im selben Blockniveau zugegriffen werden, nicht aber auf Prozeduren, die in einem tieferen Niveau liegen, oder Prozeduren, die nach der Prozedurdefinition folgen. In Listing 3.9 können Anweisungen der Prozedur `A` auf die Prozeduren `A1` und `A2` zugreifen (beispielsweise wird in Zeile 18 auf die Prozedur `A1` zugegriffen). Auf `A2A` kann nicht zugegriffen werden, da die Prozedur auf einem niedrigeren Blockniveau liegt. Die Anweisungen der Prozedur `A2A` können die Prozeduren `A`, `A1` und `A2` aufrufen, weil diese Prozeduren auf einem höheren Blockniveau liegen und vor der Prozedur `A2A` deklariert wurden.

Prozeduren können sich überdecken, indem eine Prozedur in einem niedrigeren Blockniveau denselben Namen bekommt wie eine Prozedur in einem höheren Blockniveau. In Listing 3.10 wird dies veranschaulicht.

```
1  PROCEDURE A;
2  VAR a, b, c;
3
4          PROCEDURE A;
5          CONST x = 1, y = 10, z = 6;
6
7                  PROCEDURE A1;
8                  BEGIN
9                  CALL A;
10                 END;
11
12         x := 7;
13
14 a := 1;
```

Listing 3.10: verschachtelte Prozedur mit Überdeckung

Die zweite Definition der Prozedur `A` überdeckt die erste. Der Prozeduraufruf in `A1` (Zeile 9) führt zum Aufruf der zweiten Prozedur `A`, die erste ist von `A1` nicht erreichbar.

3.1.5 Anweisungen

PL0 kennt sechs Anweisungen: die Zuweisung, die Ein- und Ausgabe, den Prozeduraufruf, die *If*-Bedingung und die *While*-Schleife. Alle Anweisungen werden mit einem Semikolon abgeschlossen. Anweisungen können durch die Schlüsselwörter `BEGIN` und `END` in einen Block zusammengefasst werden. Dabei wird kein neues Blockniveau eingeführt (siehe Abschnitt 3.1.4 auf Seite 46), sondern dient lediglich zur besseren Strukturierung und damit zu besseren Lesbarkeit.

In Tabelle 3.1 sind die einzelnen Anweisungen mit ihrer Bedeutung und Syntax aufgeführt. Ein `ausdruck` ist ein mathematischer Ausdruck, der die Zeichen $+$, $-$, $*$, $($, $)$ und / enthalten darf. Ein Ausdruck darf darüberhinaus Konstanten- und Variablennamen enthalten. Die Prioritäten der Operatoren sind in Tabelle 3.2 aufgeführt.

Anweisung	Syntax	Bedeutung
Zuweisung	`variable := ausdruck`	`variable` muss eine Variable sein, die den `ausdruck` zugewiesen bekommt.
Eingabe	`?variable`	Das Programm hält an und verlangt nach einer Benutzereingabe. Die Eingabe wird in `variable` gespeichert.
Ausgabe	`!ausdruck`	Der `ausdruck` wurde berechnet und auf einem Ausgabegerät (typischerweiser dem Bildschirm) ausgegeben.
Prozeduraufruf	`call prozedurname`	Führt die Prozedur `prozedurname` aus. Die Prozedur muss existieren.
If-Bedingung	`if bedingung then anweisung`	Die Anweisung oder der Anweisungsblock - eingeschlossen durch `BEGIN` und `END` - wird ausgeführt, wenn `bedingung` wahr ergibt.
While-Schleife	`while bedingung do anweisung`	Die Anweisung oder der Anweisungsblock - eingeschlossen durch die Schlüsselwörter `BEGIN` und `END` - wird solange ausgeführt, bis die Bedingung falsch ergibt.

Tabelle 3.1: PL0-Anweisungen

Operator	Priorität
$+, -$	geringe Priorität
$*, /$	hohe Priorität

Tabelle 3.2: Prioritäten der Operatoren in PL0

Eine `bedingung` ist ein boolescher Ausdruck. Mit ihm kann auf Gleichheit (=) und Ungleichheit (#) geprüft werden, auch sind die relationalen Operatoren <, >, <= und >= zugelassen. Durch das Schlüsselwort `ODD` kann geprüft werden, ob der folgende Ausdruck ungerade ist.

3.2 Morphemklassen

Bevor ein konkreter Syntaxbaum durch den Parser aufgebaut werden kann, muss der Lexer ein vorgegebenes PL0-Programm in Morpheme zerlegen. Die Morpheme mit ihren Klassen werden in diesem Abschnitt vorgestellt.

Zunächst werden die Schlüsselwörter betrachtet. Jedes Schlüsselwort bekommt seine eigene Morphemklasse. Den Grund kann man sich mit Hilfe der Grammatik erklären. Die Morpheme bilden in der Grammatik die Terminale. Wenn alle Schlüsselwörter genau einer Morphemklasse zugeordnet würden, müsste ein Terminal, das diese einzige Morphemklasse vertritt, als Stellvertreter für jedes Schlüsselwort dienen, was zu Worten führen würde, die kein gültiges PL0-Programm darstellen.

Dazu ein Beispiel: Angenommen die Schlüsselwörter `if` und `then` würden der Morphemklasse `reservedWord` angehören. Die Regel für die If-Anweisung würde damit folgendermaßen aussehen:

```
ifstatement = reservedWord condition reservedWord statement
```

Offensichtlich kann folgende Ableitung gefunden werden (wobei die konkreten Schlüsselwörter durch Großbuchstaben dargestellt werden):

```
IF condition THEN statement
```

Aber auch diese Ableitung wäre korrekt:

```
THEN condition THEN statement
```

Letzte Ableitung führt nicht zu einem gültigen PL0-Programm.

Für die Begrenzer `(`, `)`, `,` und `;` werden ebenfalls eigene Morphemklassen gebildet.

Die Zeichen `+` und `-` werden zu der Klasse `AOP` zusammengefasst, während die Zeichen `*` und `/` die Klasse `MOP` bilden.

Die relationalen Operatoren fallen ein wenig aus der Reihe. Die Zeichen `#`, `<`, `>`, `<=` und `>=` bilden die Klasse `RELOP`. Theoretisch könnte auch das Gleichheitszeichen `=` in diese Klasse fallen, weil es an der Stelle der anderen Zeichen stehen könnte. Allerdings wird `=` auch bei der Konstantendefinition als Zuweisungsoperator verwendet. Da an dieser Stelle kein relationaler Operator stehen darf, sondern ausschließlich das Gleichheitszeichen, muss das Zeichen einer separten Klassen zugeordnet werden. In der Grammatik muss dies berücksichtigt werden.

Die Ein- und Ausgabezeichen `?` und `!` werden in jeweils eigene Morphemklassen ausgelagert.

Übrig bleiben die Morphemklassen für die Identifizierer und die Zahlen. Ein Identifizierer besteht aus Buchstaben und Zahlen, er muss mit einem Buchstaben beginnen. Innerhalb des Identifizierers kann der Unterstrich `_` verwendet werden, sofern er immer zwischen zwei anderen Symbolen eingeschlossen ist. Eine Zahl besteht nur aus Ziffern, sie darf nicht mit Null beginnen, es sei denn es soll die Null an sich dargestellt werden.

In Tabelle 3.3 sind alle Morpheme und ihre Definition aufgelistet. Die Syntax orientiert sich dabei an der in Tabelle 3.4 dargestellten Metazeichen. Hier stehen `Digit` und `Character` für die Mengen an Ziffern und Buchstaben. Der Ausdruck `1..9` ist eine Abkürzung und bezeichnet die Ausdrücke `"1" | "2" | "3" ...`.

Morphemklasse	Elemente
jeweils eigene Klasse	`IF`, `WHILE`, `BEGIN`, `END`, `THEN`, `DO`, `CONST`, `VAR`, `PROCEDURE`, `CALL`, `ODD`
`NUMBER`	`( "1..9" {"Digit"} ) | 0`
jeweils eigene Klasse	`!`, `?`
jeweils eigene Klasse	`,`, `;`, `.`, `(`, `)`
`RELOP`	`<`, `>`, `<=`, `>=`, `#`
`EQUAL`	`=`
`AOP`	`+`, `-`
`MOP`	`*`, `/`
`IDENT`	`"Character" { {"Character"|"Digit"} [ "_" ] ("Character"|"Digit") }`

Tabelle 3.3: Morpheme von PL0

Metazeichen	Bedeutung
[...]	Ausdruck ist optional
{...}	Ausdruck kann beliebig oft wiederholt werden, kann ausgelassen werden
a | b	stellt eine Alternative zwischen den Ausdrücken `a` und `b` dar
”x”	Ausdruck `x` wird als Literal verstanden

Tabelle 3.4: Bedeutung der EBNF-Metazeichen

3.3 Konkrete Syntax

Die konkrete Syntax wurde aus [Wirth] übernommen und ist in Tabelle 3.5 dargestellt. Die Bedeutung der Metazeichen ist in Tabelle 3.4 dargestellt.

Die konkrete Syntax soll nicht näher erläutert werden, da die Grammatik selbst erklärend ist.

PL0 wird im Laufe dieser Arbeit mit der Erweiterung SableCC's und mit dem Werkzeug *JastAdd* implementiert. Die Grammatikspezifikationen für diese Werkzeuge weichen etwas von der hier dargestellten Form ab. Insbesondere werden die Morpheme in den Grammatikspezifikationen nur mit ihren Klassennamen ausgedrückt.

3.4 Abstrakte Syntax

Die abstrakte Syntax beschreibt die Struktur der Sprache. Da Konzepte wie “enthält” und “besteht aus” in UML-Klassendiagrammen genutzt werden können, kann die abstrakte Syntax durch diese dargestellt werden. Für die Sprache PL0 wurden vier Hauptkonzepte eingeführt:

- *Execution-Unit* - eine ausführbare Einheit
- *Statement* - eine Anweisung im Programm
- *Expression* - ein mathematischer Ausdruck
- *Condition* - eine Bedingung für die `If`- und `While`-Anweisungen

program	=	block ”.”
block	=	[”CONST” ident ”=” number {”,” ident ”=” number} ”;”] [”VAR” ident { ”,” ident} ”;”] { ”PROCEDURE” ident ”;” block ”;”} statement
statement	=	[ident ”:=” expression | ”CALL” ident | ”?” ident | ”!” expression | ”BEGIN” statement {”;” statement} ”END” | ”IF” condition ”THEN” statement | ”WHILE” condition ”DO” statement]
condition	=	”ODD” expression | expression (”=”| ”# ” | ”<” | ”<=” | ”>” | ”>=”) expression
expression	=	[”+” | ”-”] term {(”+” | ”-”) term}
term	=	factor {(”∗” | ”/”) factor}
factor	=	ident | number | ”(” expression ”)”

Tabelle 3.5: Konkrete Syntax von PL0 in EBNF Darstellung

Diese vier Konzepten sollen in den nächsten Abschnitten näher vorgestellt werden.

3.4.1 Das Konzept Execution-Unit

Das Konzept ist in Abbildung 3.1 dargestellt.

Die Klasse `ExecutionUnitAS` stellt eine ausführbare Einheit in einem PL0-Programm dar. Sie kann Variablen, Konstanten und Prozeduren besitzen, muss aber mindestens eine Anweisung (siehe Klasse `StatementAS`) beinhalten.

Die Klasse `ExecutionUnitAS` ist abstrakt. Sie besitzt die beiden Unterklassen `ProgramAS` und `ProcedureAS`. Die Klasse `ProgramAS` repräsentiert ein PL0-Programm, das keinen Namen hat. Dies unterscheidet es von der Prozedur, die die Schnittstelle `NameableAS` implementiert, und demzufolge einen Namen besitzt.

Eine Sonderstellung nimmt die Klasse `BlockContainerAS` ein. Sie gehört eigentlich nicht zur abstrakten Syntax, wird aber während des Parsens benötigt, um Variablen, Konstanten und Prozeduren zwischenzuspeichern. Das resultierende PL0-Modell (oder abstrakter Syntaxgraph/-modell) enthält keine Instanz der Klasse `BlockContainerAS`.

3.4.2 Das Konzept Statement

Das Konzept *Statement* ist in Abbildung 3.2 dargestellt.

Es stellt eine einzelne Anweisung dar, die ausgeführt werden kann. PL0 kennt fünf grundlegende Anweisungen: Ein- und Ausgabe, die Zuweisung, den Prozeduraufruf und eine Anweisung mit Bedingung. Im Klassendiagramm findet man die entsprechenden Klassen wieder. Eine Ausgabeanweisung zeigt auf einen Ausdruck, die Eingabe wird in eine Variable eingelesen, die Zuweisung weist einer Variablen einen Ausdruck zu, ein Prozeduraufruf muss die aufzurufende Prozedur kennen und die Bedingungsanweisung muss über eine Bedingung und eine Anweisung verfügen.

Eine Sonderrolle spielt die Klasse `StatementsAS`. Sie besitzt mindestens eine Anweisung. Durch diese Klasse ist es möglich, einer Prozedur oder einer Bedingungsanweisung (im Fall der wahren Bedingung) mehrere Anweisungen zu geben. In der konkreten Syntax spiegelt sich dieses durch die Blöcke wieder, die mit den Schlüsselwörtern `BEGIN` und `END` eingerahmt sind.

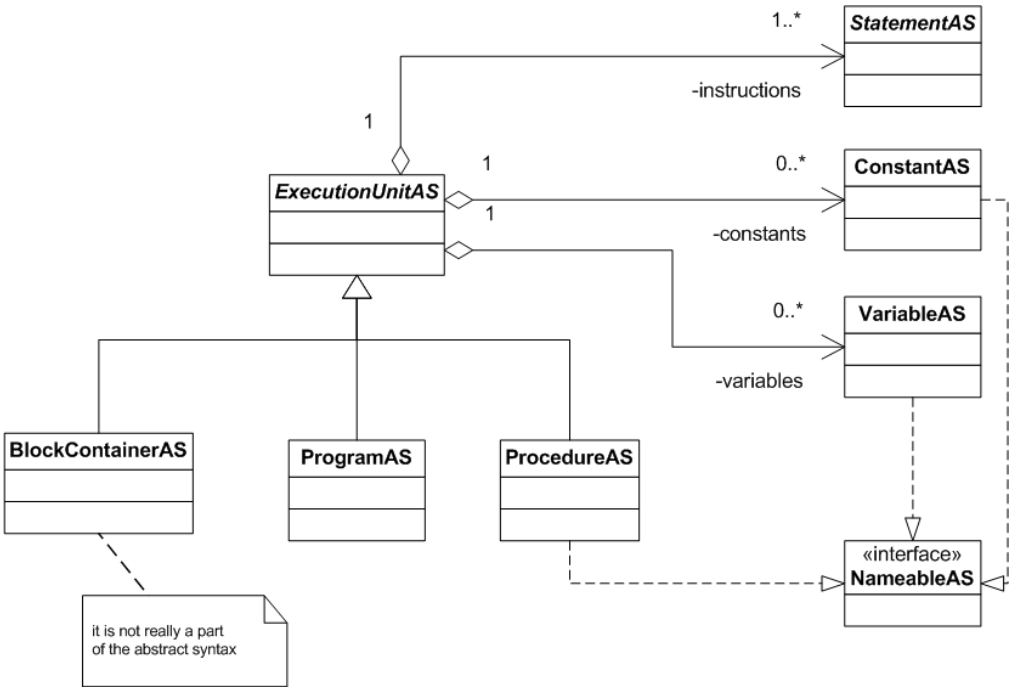


Abbildung 3.1: Konzept *ExcecutionUnit*

3.4.3 Das Konzept Expression

Das Konzept Expression ist in Abbildung 3.3 dargestellt.

Es repräsentiert mathematische Ausdrücke in Form eines Baumes (vgl. *Composite*-Muster [GOF]). Ein Blatt des Baumes kann eine Variable (Klasse `VariableAS`), eine Konstante (Klasse `ConstantAS`) oder ein Zahlenliteral (Klasse `NumberAS`) sein. Ein innerer Knoten des Baumes kann eine binäre Operation (Klasse `BinaryOperatorAS`) oder eine unäre Operation (Klasse `UnaryOperatorAS`) sein. Instanzen der Operationsklassen besitzen einen Typen, wobei der Typ angibt, um welche Operation es sich handelt. Zu beachten ist, das die unäre Operation nur die Operationen *plus* und *minus* als Typ zulässt.

Die Prioritäten der Operationen werden indirekt über die Baumstruktur gebildet. In der konkreten Syntax werden die Prioritäten über das *semantische Niveau* definiert. Innerhalb einer Grammatik für mathematische Ausdrücke ergeben sich unterschiedliche Stufen, die als *semantische Niveaus* bezeichnet werden. Dabei legt das semantische Niveau die Priorität des Operators, der auf dieser Stufe eingefügt wird, fest. Operatoren auf einem hohen semantischen Niveau besitzen eine kleinere Priorität als ein Operator auf einem tieferem Niveau. Dazu ein kleines Beispiel. Betrachtet werde die folgende Grammatik:

$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow T * F \end{aligned}$$

In dieser Grammatik gibt es zwei Stufen, damit auch zwei semantische Niveaus.

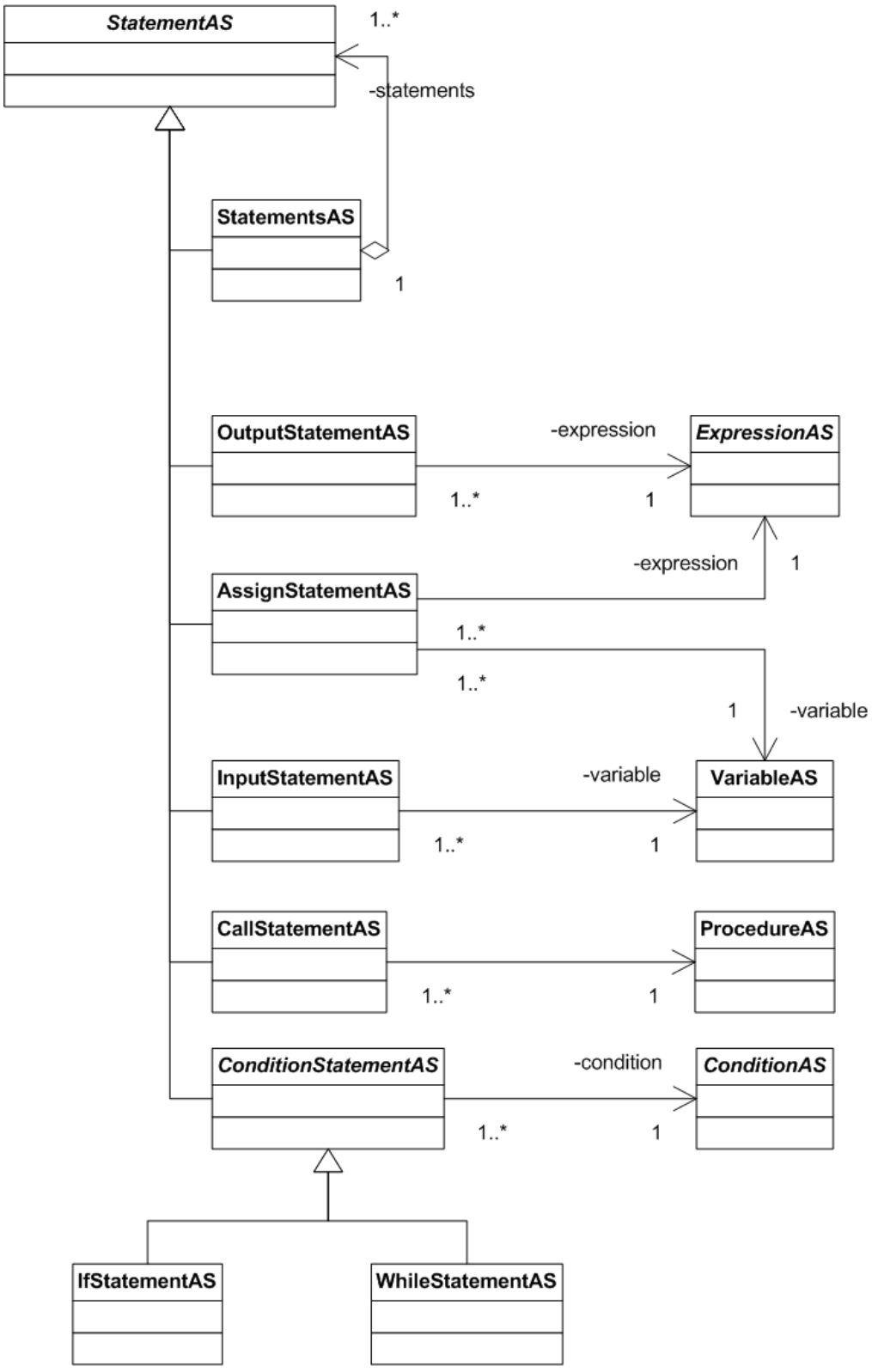


Abbildung 3.2: Konzept *Statement*

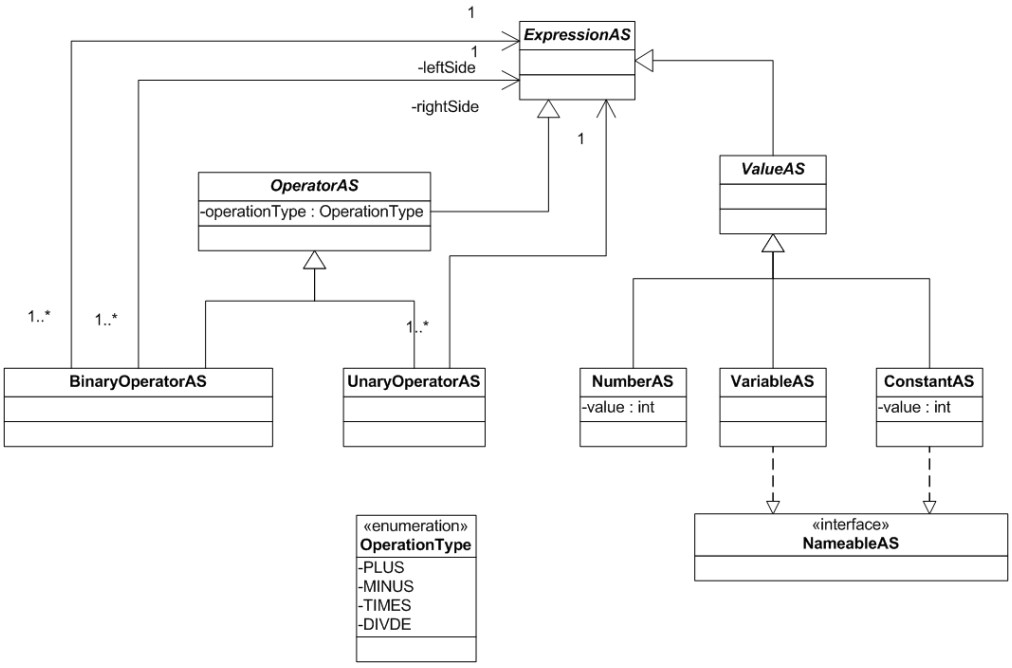


Abbildung 3.3: Das Konzept *Expression*

Da der Additionsoperator auf einem höheren Niveau liegt, besitzt er eine kleinere Priorität als der Multiplikationsoperator.

Die implizierte Priorisierung soll anhand zweier Beispiele erläutert werden. Der mathematische Ausdruck $4 + 5 * 3$ ergibt den in Abbildung 3.4 dargestellten Baum.

Der Ausdruck $(4 + 5) * 3$ wird durch den in Abbildung 3.5 dargestellten Baum repräsentiert.

3.4.4 Das Konzept Condition

Das Konzept *Condition* ist in Abbildung 3.6 dargestellt.

In PL0 gibt es zwei Arten von Bedingungen: die relationale (vgl. Klasse `ReloOpConditionAS`), in der zwei mathematische Ausdrücke verglichen werden, und die Prüfung, ob sich ein mathematischer Ausdruck zu einer ungeraden Zahl auswerten lässt (vgl. Klasse `OddConditionAS`). Innerhalb der Klasse `RelOpConditionAS` wird vermerkt, um was für eine relationale Operation es sich handelt.

3.4.5 Die Schnittstelle NameableAS

Das Klassendiagramm der Schnittstelle `Nameable` ist in Abbildung 3.7 dargestellt. Die Schnittstelle bezeichnet in PL0 ein Konstrukt, das benannt ist (oder in der konkreten Syntax einen Identifizierer besitzt). Benannt werden können Prozeduren - sie werden mit `call` aufgerufen -, Variablen und Konstanten, die in Ausdrücken verwendet werden dürfen. Diese Schnittstelle wird benötigt, um einen einfacheren Zugriff auf die benannten Objekte zu erhalten (siehe Abschnitt 3.5 auf Seite 56).

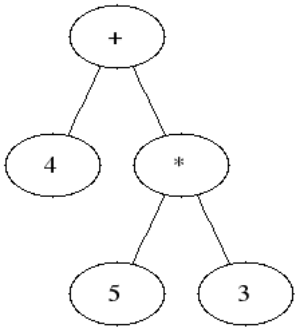


Abbildung 3.4: Baum für den Ausdruck $4 + 5 * 3$ mit implizierter Priorität

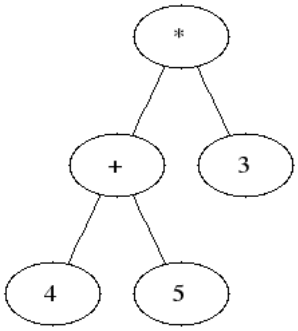


Abbildung 3.5: Baum für den Ausdruck $(4 + 5) * 3$

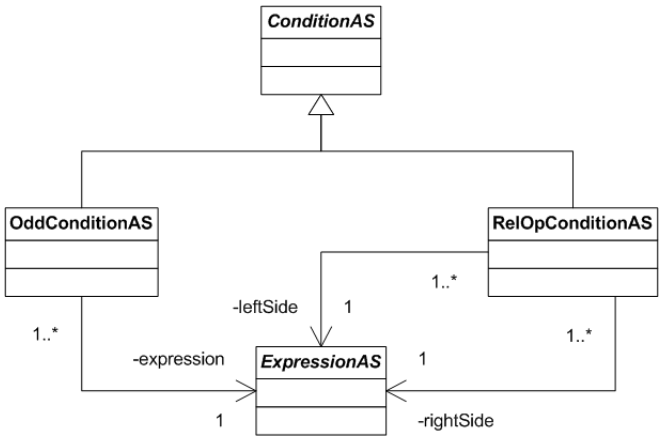


Abbildung 3.6: Das Konzept *Condition*

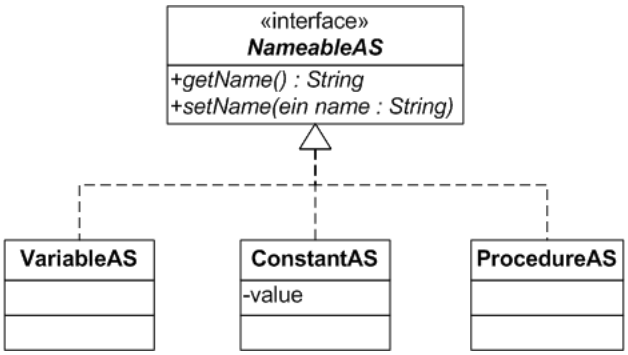


Abbildung 3.7: Die Schnittstelle *NameableAS*

3.5 Die semantische Analyse

Mit Hilfe der semantischen Analyse ist es möglich kontextsensitive Informationen zu verarbeiten. Zur Erinnerung: alle gängigen Parserverfahren basieren auf kontextfreien Grammatiken. Eine Einbeziehung des Kontextes ist dabei nicht möglich.

Auch kann man aus einem konkreten Syntaxbaum ein abstraktes Modell (bzw. abstrakter Syntaxgraph/-modell) generieren. Die semantische Analyse wird im folgenden als Attributauswertung (siehe Abschnitt 2 auf Seite 19) verstanden. Dieser Abschnitt besteht aus zwei Teilen: zum einen wird gezeigt, welche Schritte notwendig sind, um die semantischen Regeln von PL0 umzusetzen, und zum anderen wird gezeigt, wie aus dem konkreten Syntaxbaum ein abstraktes Modell generiert werden kann.

3.5.1 Implementierung der semantischen Regeln

In Kapitel 3.1 auf Seite 43 wurde ausführlich die Semantik von PL0 informell beschrieben. Diese gilt es in diesem Kapitel umzusetzen. Für die Umsetzung ist das Konzept der *Umgebung* (im Englischen *Environment*) notwendig. Dieses soll zunächst erklärt werden.

Variablen, Konstanten und Prozeduren werden an irgendeiner Stelle im Programm deklariert, das heißt, Variablen, Konstanten und Prozeduren werden Namen in Form von Identifizierern zugeordnet. Diese können an einer anderen Stelle im Programm verwendet werden. Nun ist für den Computer nicht klar, welcher Identifizierer was repräsentiert, für ihn ist der Identifizierer lediglich eine Zeichenkette. Es müssen den Zeichenketten eine Bedeutung zugeordnet werden. In dem Ausdruck `call proc` stellt die Zeichenkette `proc` eine Prozedur dar, da nur Prozeduren mit `call` aufgerufen werden können. Das heißt, an der Stelle, an der die Prozedur deklariert wird, muss sich gemerkt werden, dass es sich bei diesem Identifizierer um eine Prozedur handelt. Das wird über die Umgebung realisiert. In ihr wird vermerkt, welche Identifizierer Variablen, Konstanten und Prozeduren entsprechen. Somit kann der Identifizierer `proc` als Prozedur erkannt werden, sofern `proc` tatsächlich eine zuvor deklarierte Prozedur bezeichnet. Wenn die Analyse gelingt, kann mit dem Identifizierer `proc` die Prozedur verknüpft werden, den er repräsentiert.

Die Umgebung muss ein paar Methoden bereitstellen, die in Tabelle 3.6 zusammengefasst sind. In dieser Tabelle tritt die Schnittstelle `NameableAS` in Erscheinung.

Methodensignatur	Beschreibung
`lookup(String ident):NameableAS`	Sucht nach dem Identifizierer `ident` in den Variablen, Konstanten und Prozeduren. Wenn der Identifizierer nicht gefunden wurde, wird die darüberliegende Umgebung durchsucht. Falls der Identifizierer nicht in der obersten Umgebung zu finden ist, wird `null` zurückgegeben. Falls der Identifizierer gefunden wird, wird die entsprechende Variable, Konstanten oder Prozedur zurückgegeben
`lookupLocal(String ident):NameableAS`	Verhält sich wie `lookup`, schaut aber lediglich in der aktuellen Umgebung nach, die darüberliegenden werden nicht durchsucht
`addVariable(VariableAS var)`	Fügt die Variable `var` in die aktuelle Umgebung ein. Falls sie schon vorhanden ist, wird eine Fehlermeldung ausgegeben.
`addConstant(ConstantAS constant)`	Verhält sich wie `addVariable`, allerdings wird der Umgebung eine Konstante hinzugefügt.
`addProcedure(ProcedureAS proc)`	Verhält sich wie `addVariable`, allerdings wird der Umgebung eine Prozedur hinzugefügt.
`setPredecessor(Environment env)`	Setzt den Vorgänger dieser Umgebung.

Tabelle 3.6: Methoden der Umgebung

Diese ermöglicht das Nachsehen der Zeichenkette mit einheitlichem Rückgabewert. Eine Alternative wäre für jedes bezeichnete Konstrukt (Variable, Konstante oder Prozedur) eine eigene Methode einzufügen, die entweder ein Konstrukt zurückgibt oder `null`, falls es kein Konstrukt mit dem Identifizierer gibt.

Im folgenden werden die semantischen Regeln für die Prozeduren, Konstanten und Variablen vorgestellt.

3.5.2 Semantische Regeln

In diesem Abschnitt soll der Rahmen für die semantische Regeln gebaut werden. Die semantischen Regeln an sich befinden sich in Anhang A auf Seite 133. Hier wird lediglich erläutert, welche Attribute für die Berechnung notwendig sind.

Attribute der Grammatiksymbole

Um eine Attributierung für eine Grammatik aufzustellen, bedarf es einiger Attribute. In diesem Fall reichen genau drei. Alle Nichtterminale erhalten die Attribute *ast* und *env*, wobei *ast* für eine Instanz aus der abstrakten Syntax und *env* für die Umgebung steht. Die Terminale erhalten das Attribut *value*, was den Wert des Morphems speichert (diese werden vom Lexer gesetzt).

Das Attribut *env* ist ein ererbtes Attribut, was bedeutet, dass es im Baum von oben nach unten gereicht wird. Dagegen sind die Attribute *ast* und *value* sog. synthetisierte Attribute, die im Baum von unten nach oben wandern (siehe Kapitel 2.3, Seite 31).

Jedem Attribut kann ein Typ zugeordnet werden, auf den sich in den semantischen Regeln bezogen werden kann. Das Attribut *env* erhält den Typ `Environment` und das Attribut *value* den Typ `String`. Das Attribut *ast* besitzt eine Sonderrolle. Grundsätzlich kommen für dieses Attribut alle Typen der abstrakten Syntax in Frage (beispielsweise `ConditionAS`, `CallStatementAS`, ...). Der Rückgabetyp ist abhängig von dem Nichtterminal, das gerade betrachtet wird. Für alle Nichtterminale sind in Tabelle 3.7 die zugehörigen Typen aufgeführt.

Die spitzen Klammern stehen für die *Generics* von Java. So bedeutet `List<VariableAS>`, das es sich dabei um eine Liste von Instanzen der Klasse `VariableAS` handelt. In der Tabelle fällt auf, dass fast alle verwendeten Typen abstrakt sind (wie beispielsweise `ConditionAS` oder `ExpressionAS`). Die konkreten Typen werden in den im Anhang A dargestellten semantischen Regeln gebaut.

Nichtterminal	**Typ für *ast*-Attribut**
Program	`ProgramAS`
Block	`BlockContainerAS`
Decl	`BlockContainerAS`
ConstVarDecl	`BlockContainerAS`
ConstDecl	`List<ConstantAS>`
ConstAssign	`List<ConstantAS>`
VarDecl	`List<VariableAS>`
VarEnum	`List<VariableAS>`
ProcedureDecl	`List<ProcedureAS>`
Statement	`StatementAS`
StatementEnum	`List<Statement>`
Condition	`ConditionAS`
Expression	`ExpressionAS`
Term	`ExpressionAS`
Factor	`ExpressionAS`

Tabelle 3.7: Nichtterminale und ihre entsprechenden Typen

4 SableCC

Die Erstellung eines Parsers gestaltet sich als sehr aufwendig, wenn alle Schritte - lexikalische, syntaktische und semantische Analyse - von Hand implementiert werden. Man hat festgestellt, dass sowohl die lexikalische als auch die syntaktische Analyse automatisiert werden können. Aus dieser Erkenntnis wurden sog. *Compiler-Compiler* entwickelt. Diese sind in der Lage anhand einer formalen Beschreibung einen Lexer und einen Parser zu generieren. Für die lexikalische Analyse werden die Morpheme (im Englischen *Token*) mittels regulärer Ausdrücke beschrieben. Aus diesen Ausdrücken werden einzelne reguläre Automaten generiert, die zusammengesetzt den Lexer ergeben, der in der Lage ist, einen Eingabetext in einen Strom von Morphemen zu transformieren.

Für die syntaktische Analyse wird eine kontextfreie Grammatik benötigt. Diese dient als Eingabe für den Parser-Generator, der einen Strom von Morphemen in einen konkreten Syntaxbaum transformiert.

Um den konkreten Syntaxbaum zu erzeugen, gibt es verschiedene Verfahren (siehe Abschnitt 2.2.2, Seite 25).

In diesem Kapitel wird zunächst der Compiler-Compiler SableCC vorgestellt. Im Rahmen der Diplomarbeit von [Kon] wurde SableCC um verschiedene Konstrukte erweitert. Im zweiten Teil dieses Kapitels wird diese Erweiterung vorgestellt.

Alle Beispiele beziehen sich auf die Sprache PL0 (siehe Kapitel 3 auf Seite 43).

4.1 Originales SableCC

SableCC wurde im Rahmen der Diplomarbeit von [GAG] entwickelt. Es generiert sowohl den Lexer als auch den Parser für Sprachen der Klasse LALR(1). SableCC wurde in der Programmiersprache Java implementiert. Lexer und Parser werden ebenfalls in der Sprache Java generiert. Im Gegensatz zu anderen Compiler-Compiler wie beispielsweise AntLr[Antlr] oder Beaver[Beaver] ist es nicht möglich innerhalb der Spezifikation Quelltext einzufügen. Dies geschieht mittels eines Frameworks, das ebenfalls generiert wird (siehe Abschnitt 4.1.2). Darüber hinaus wird eine Klassenhierarchie für die Knoten des konkreten Syntaxbaums erzeugt. Während der Konstruktion des SableCC-Compiler-Compiler wurde viel Wert auf den Einsatz des objektorientierten Paradigmas und von Entwurfsmustern gelegt.

In diesem Abschnitt wird SableCC näher vorgestellt. Zunächst wird die Spezifikation betrachtet, die für die Generierung notwendig ist. Anschließend wird die Generierung des Frameworks erläutert. Am Schluss werden die Nachteile von SableCC aufgezeigt, die im Rahmen von [Kon] entdeckt wurden und die dazu geführt haben, SableCC zu erweitern.

4.1.1 SableCC-Spezifikation

Eine SableCC-Spezifikation besteht aus mehreren Abschnitten, die jeweils durch ein Schlüsselwort eingeleitet werden. Diese Abschnitten werden im Folgenden näher vorgestellt.

Syntax	Bedeutung
'a'	ein einzelnes Zeichen
a | b | c	die Auswahl zwischen mehren Teilausdrücken
a*	beliebig oft
a+	mindestens einmal
a?	höchstens einmaliges Vorkommen
['1'.. '9']	ein Bereich

Tabelle 4.1: Syntax der regulären Ausdrücke

Abschnitt `package`

Mit dem Schlüsselwort `package` wird dem Generator gesagt, in welches Paket der Parser generiert werden soll. Alle erzeugten Klassen finden sich in diesem angegebenen Paket wieder. Die Syntax entspricht derjenigen von Java, das heißt, die einzelnen Bestandteile sind durch den Punkt voneinander getrennt. Der Paketname wird durch ein Semikolon abgeschlossen.

Abschnitt `Helpers`

Mit dem Schlüsselwort `Helpers` wird ein Abschnitt eingeleitet, in dem Hilfsdefinitionen stehen können, die im Morphemabschnitt verwendet werden können. So können hier Abkürzungen für wiederkehrende reguläre Ausdrücke definiert werden. Dieser Abschnitt ist nicht obligatorisch. Zu beachten ist, dass diese Regeln keine Rekursion beinhalten dürfen, das heißt, dass das Symbol der linken Seite nicht auf der rechten vorkommen darf. Die Regeln besitzen folgenden Aufbau:

```
Bezeichner = regulärer Ausdruck ;
```

Jede Regel besitzt einen zu definierenden Bezeichner. Diesem wird ein regulärer Ausdruck zugewiesen. Jede Regel endet mit einem Semikolon. Die Syntax und ihre Bedeutung für die regulären Ausdrücke ist in Tabelle 4.1.1 dargestellt.

Abschnitt `Tokens`

Mit dem Schlüsselwort `Tokens` wird die Morphemdefinition eingeleitet. Sie ähnelt im Aufbau dem Abschnitt der Hilfsdefinitionen, wobei nun rekursiven Regeln zugelassen sind. Auch können natürlich die Hilfsdefinitionen verwendet werden. In Listing 4.1 ist ein Beispiel für die Morphemdefinition der relationalen, mathematischen Zeichen und der Zahlen aus PL0 gegeben.

```
relop = ('<=' | '>=' | '<' | '>' | '#');
aop = ('+' | '-');
mop = ('*' | '/');
number = ['1'..'9']['0'..'9']* | '0';
```

Listing 4.1: Beispiel für die Morphemdefinition in SableCC

Das Morphem `relop` repräsentiert die relationalen Operatoren, wobei das Zeichen `#` für das Ungleich steht. Mit `aop` und `mop` werden die mathematischen Operatoren für Addition und Multiplikation definiert. Das Morphem `number` repräsentiert alle natürlichen Zahlen.

Die Reihenfolge, in der die Morpheme definiert werden, ist relevant. Allgemeine Definitionen sollten weiter unten gesetzt werden. Dazu ein Beispiel: in PL0 gibt es das Schlüsselwort `IF` und es gibt Identifikatoren. Die Definition der Identifikatoren schließt die Definition des Schlüsselwortes `IF` ein (siehe Listing 4.2, in dem `letter` für einen Buchstaben und `digit` für eine Zahl steht).

```
if = 'IF';
ident = letter (letter | digit)*
  ('_' letter (letter | digit)*)*;
```

Listing 4.2: Beispiel für die Morphemdefinition IF und Identifikator

Das Schlüsselwort `IF` kann also durch die Definition des Identifikators gebildet werden. Wenn nun ein `IF` gelesen wird, wird es mit allen Definitionen der Reihe nach verglichen. Da die IF-Definition vor dem Identifikator steht, wird diese genutzt (und das Morphem `if` erkannt). Falls die Definition des Identifikators vor der `IF`-Definition stehen würde, würde diese benutzt werden (das Morphem `ident` würde erkannt werden). Einen interessanten Fall gilt es zu beachten: im Quellprogramm kann ein Identifikator stehen, der mit einem Schlüsselwort beginnt, beispielsweise `IFTrue`. In diesem Fall wird nach dem Prinzip der längsten Kette vorgegangen, das heißt, es wird versucht eine Regel zu finden, die das gesamte Morphem abdeckt. Im Beispiel wäre dies die Identifikatordefinition und nicht die Definition des `IF`-Schlüsselwortes.

Abschnitt `Ignored Tokens`

Nach der Morphemdefinition können Morpheme definiert werden, die ausgelassen werden sollen (sie werden dem Parser nicht übergeben). Dazu gehören Zeichen wie Tabulatoren, Leerzeichen, Zeilenumbrüche etc. Dieser Abschnitt wird mit dem Schlüsselwort `Ignored Tokens` eingeleitet und darf nur Morpheme enthalten, die im Abschnitt `Tokens` definiert wurden. Abgeschlossen wird dieser Abschnitt mit einem Semikolon.

Abschnitt `Productions`

Auf den Abschnitt der Morpheme bzw. der ignorierbaren Morpheme folgen die Regeln der Grammatik, eingeleitet durch das Schlüsselwort `Productions`. Eine Solche Produktion/Regel besteht aus einer linken Seite, einem Gleichheitszeichen, und einer rechten Seite. Abgeschlossen wird jede Regel mit einem Semikolon. Auf der linken Seite darf genau ein Nichtterminal stehen (zur Erinnerung: SableCC verarbeitet nur kontextfreie Grammatiken). Die Nichtterminale müssen als solche nicht gekennzeichnet werden. Es wird nur gefordert, dass sie sich von den Morphemnamen unterscheiden. Auf der rechten Seite kann ein komplexer Ausdruck stehen: er kann Terminale, Nichtterminale und mehrere Regelalternativen enthalten. Dazu kommen noch die bekannten Symbole der regulären Ausdrücke (siehe Tabelle 4.1.1), die in diesem Fall eine ähnliche Bedeutung haben. Um dies zu verdeutlichen, wird die Regel der Bedingung aus PL0 betrachtet (siehe Listing 4.3).

```
condition = {odd_condition} odd expression
 | {relop_condition} [left_side]:expression relop
        [right_side]:expression;
```

Listing 4.3: Beispiel Condition-Regel

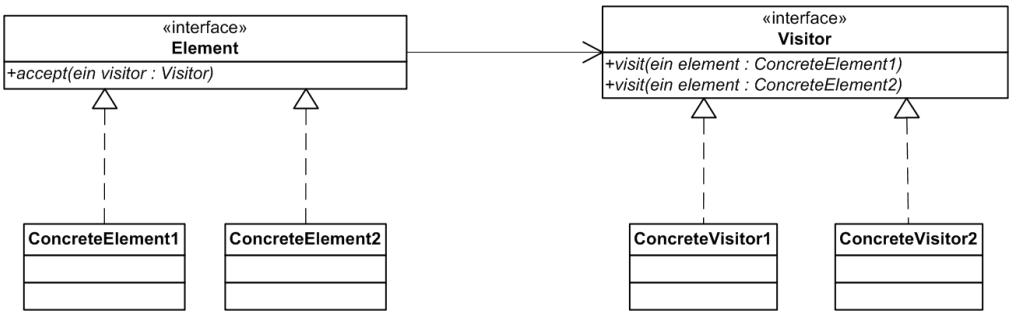


Abbildung 4.1: Besuchermuster nach [GOF]

Die Regel definiert das Nichtterminal `condition` anhand zweier möglicher Ausprägungen. Diese Ausprägungen werden im weiteren Verlauf *Alternativen* genannt. Jede Alternative besitzt einen Namen, der der Alternative in geschweiften Klammern ({ und }) vorangestellt wird. In diesem Fall gibt es zwei Alternativen mit den Namen `odd_condition` und `relop_condition`. Die Alternativen werden durch einen senkrechten Strich | voneinander getrennt. Eine Alternative kann aus Terminalzeichen und Nichtterminalzeichen bestehen. In diesem Beispiel setzt sich die Alternative `relop_condition` aus dem Terminal `relop` und zwei Nichtterminalen `expression` zusammen. Da das Nichtterminal `expression` zweimal verwendet wird, muss jeder Verwendung einen Namen gegeben werden. Dieser Name steht vor dem Grammatiksymbol (Terminal oder Nichtterminal) in eckigen Klammern ([und]) und wird mit einem Doppelpunkt abgeschlossen. Um die Definition von Regeln zu vereinfachen, können die Zeichen aus den regulären Ausdrücken verwendet werden (siehe Tabelle 4.1.1).

4.1.2 Framework

SableCC generiert aus der Spezifikation neben dem Lexer und dem Parser ein Framework. Wie oben erwähnt, lässt sich kein Quelltext in die Spezifikation einfügen. Statt dessen werden Klassen abgeleitet und implementiert, um eigenen Quelltext einzubringen.

Das generierte Framework basiert auf dem Besuchermuster (im Englischen *Visitor-Pattern*) nach [GOF]. Dieses Muster ist in Abbildung 4.1 dargestellt.

Das Besuchermuster wurde so konzipiert, dass ein neuer Besucher (in Abbildung 4.1 auf der rechten Seite dargestellt) hinzugefügt werden kann, ohne die bestehende Klassenhierarchie anpassen zu müssen. Der Autor von SableCC hat sich dagegen zum Ziel gesetzt, neue Elemente (vgl. Schnittstelle `Element`) in die Klassenstruktur aufzunehmen, ohne die bisherige Klassenstruktur ändern zu müssen. Um dieses Ziel zu erreichen, wurde das Besuchermuster modifiziert (siehe Abbildung 4.2). Die Namen der Elemente wurden geändert (siehe Tabelle 4.2).

Die Schnittstelle `Switch` (vgl. Schnittstelle `Visitor`) enthält nun keine Methoden mehr. Stattdessen erben andere Schnittstellen von ihr, die die `case`-Methoden definieren. Wenn nun Elemente in die Elementhierarchie hinzugefügt werden sollen, wird eine neue Schnittstelle mit den entsprechenden `case`-Methoden erzeugt (in Abbildung 4.2 trifft dies auf die Klasse `ConcreteElement3` zu, dabei wird die Schnittstelle `Analysis2` hinzugefügt). Anschließend wird eine neue Schnittstelle `AllAnalysis` hinzugefügt, die

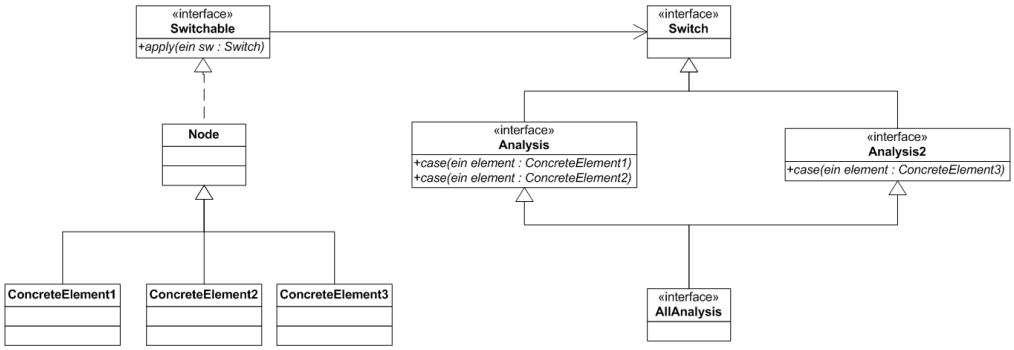


Abbildung 4.2: Erweitertes Besuchermuster in SableCC

Originales Besuchermuster	**Besuchermuster in SableCC**
Schnittstelle `Visitor`	Schnittstelle `Switch`
Schnittstelle `Element`	Schnittstelle `Switchable`
Methode `visit`	Methode `case`
Methode `accept`	Methode `apply`

Tabelle 4.2: Umbenennungsschema für das Besuchermuster

von allen Visitor-Schnittstellen erbt. So wird die bestehende Klassenhierarchie nicht angetastet.

4.1.3 Generierungsregeln für das Framework

Das Framework wird mit Hilfe der Informationen aus der Spezifikation generiert. Dabei kann zwischen zwei Teilaspekten unterschieden werden. Zum einen werden die Terminale und Nichtterminale Teile der Elementhierarchie des Besuchermusters. Zum anderen finden sich die Grammatikregeln in den Methoden der Besucherschnittstellen wieder. Im Folgenden sollen die Regeln im einzelnen betrachtet werden.

Terminale und Nichtterminale

Es wird eine abstrakte Klasse `Node` generiert, die die Basis für die Terminal- und Nichtterminalklassen ist. Sie implementiert die Schnittstelle `Schwitchable`, die die Methode `apply` definiert. Für jedes Grammatiksymbol (Terminal und Nichtterminal) wird eine Klasse generiert. Von der Klasse `Node` erben alle Klassen der Nichtterminale. Zusätzlich erben die Klassen `Token` und `Start` von der Klasse `Node`. Die Klasse `Token` bildet die Basisklasse für alle Terminale. Die Klasse `Start` dagegen bildet eine Ausnahme. Diese repräsentiert das Startsymbol. An dieser Klasse hängt nach dem Parservorgang der gesamte konkrete Syntaxbaum. In Abbildung 4.3 ist die Klassenstruktur veranschaulicht (für das folgende Beispiel).

Die Namen der erzeugten Klassen für die Terminale beginnen mit einem `T` und enden auf dem Namen des Terminals, wobei zu beachten ist, dass der Anfangsbuchstabe eines jeden Terminals als Großbuchstabe in den Klassennamen eingeht. Für jedes Nichtterminal wird eine abstrakte Klasse generiert, die mit einem `P` beginnt und mit dem

Namen des Nichtterminals endet, wobei auch hier wieder zu beachten ist, dass der erste Buchstabe des Nichtterminals als Großbuchstabe in die Namensbildung eingeht. Die abstrakten Klassen der Nichtterminale bilden die Basisklassen für die weiteren Klassen, die im nächsten Abschnitt besprochen werden.

Grammatikregeln

Eine Grammatikregel besteht aus mindestens einer Alternativen. Für jede Alternative wird eine Klasse generiert, deren Namen mit einem `A` beginnt, den Namen der Alternativen als Infix trägt und mit dem Namen des Nichtterminals auf der rechten Seite abgeschlossen wird. Die Alternativklassen erben von der abstrakten Klasse, die aus dem Nichtterminal der linken Seite hervorgeht. Bei der Namensgebung der Alternativen und der Regeln innerhalb der Spezifikation ist darauf zu achten, dass nur Kleinbuchstaben verwendet werden dürfen. Möchte man die Namen übersichtlicher gestalten, kann der Unterstrich _ verwendet werden. Dieser wird im Klassennamen gelöscht und der darauffolgende Buchstabe wird groß geschrieben. Zur Veranschaulichung wird das folgende Beispiel verwendet (siehe Listing 4.4). Es handelt sich dabei um einen Ausschnitt aus der PL0-Grammatik (siehe Abschnitt 3.3 auf Seite 50), der die Bedingung repräsentiert. Die Bedingung besteht aus zwei Alternativen `odd_condition` und `relop_condition`. Für beide Alternativen wird jeweils eine Klasse generiert, die die folgenden Namen tragen: `AOddConditionCondition` und `ARelopConditionCondition`. Das zweite Vorkommen von `Condition` im Namen ergibt sich durch das Nichtterminal auf der linken Seite der Regel.

```
condition = {odd_condition} odd expression
 | {relop_condition} [left_side]:expression relop
         [right_side]:expression;
```

Listing 4.4: Beispiel Condition-Regel

In Abbildung 4.3 findet sich beispielhaft eine kleine Klassenhierarchie wieder, die für die Bedingung generiert wird.

Besucher

Bisher wurde lediglich die Elementhierarchie des Besuchermusters betrachtet. SableCC generiert ebenfalls die eigentliche Besucherhierarchie. Dies verläuft wie folgt: Es wird eine Schnittstelle `Switch` generiert, die keine Methode enthält. Außerdem wird eine Schnittstelle `Analysis` generiert, die für jede Regelalternative und für jede Morphemdefinition eine Methode enthält. Der Methodenname setzt sich aus dem Präfix `case` und dem Namen zusammen, der sich auch bei der Elementhierarchie ergibt (für die Morpheme `T`*Morphemname* bzw. `A`*AlternativRegelName*). Die Signatur der `case`-Methode ist folgende:

```
void caseAName(AName node);
```

Die Klasse `AnalysisAdapeter` implementiert die Schnittstelle `Analysis`, die alle `case`-Methoden implementiert. Diese Implementierung bleibt allerdings leer. Es werden zwei Referenzen `in` und `out` vom Typ `Hashtable<Node,Object>` hinzugefügt, auf die mit den Methoden `getIn(...)`, `setIn(...)`, `getOut(...)` und `setOut(...)` zugegriffen werden kann. Hier können Objekte während des Durchlaufens des konkreten Syntaxbaumes abgelegt werden. Die Klassen `DepthFirstAdapter` und `ReversedDepthFirstAdapter` leiten die Klasse

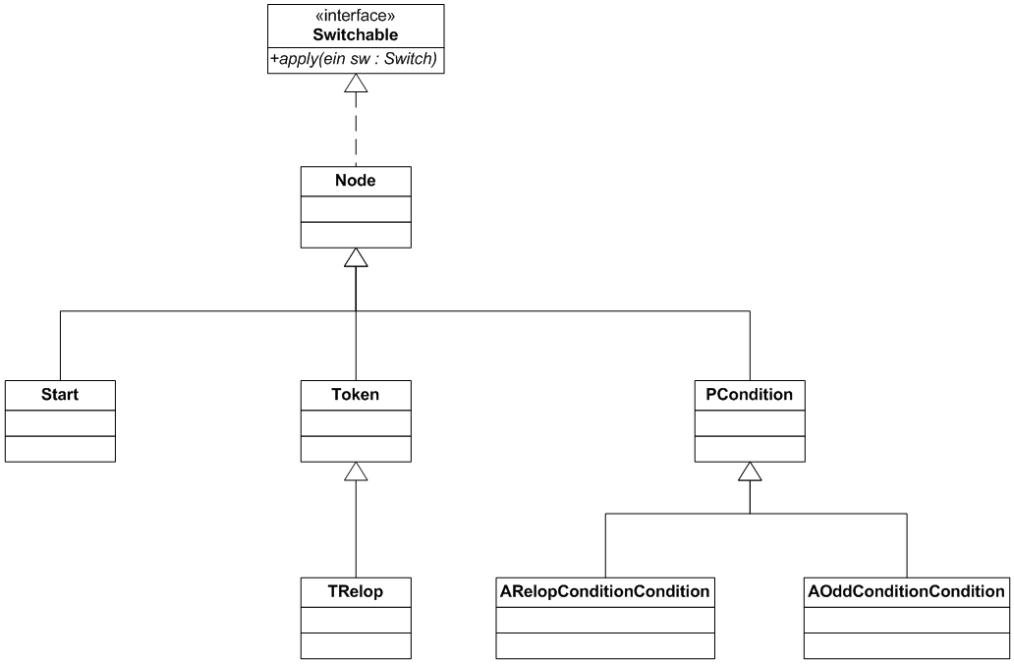


Abbildung 4.3: Elementhierarchie SableCC

`AnalysisAdapter` ab und implementieren eine Traversierung durch den Baum (einmal von links nach rechts und von rechts nach links). Beide Klasse enthalten für jede `case`-Methode jeweils zwei weitere Methoden: `inAName(AName node)` und `outAName(AName node)`, wobei `Name` für den Postfix der `case`-Methode steht. Diese Methoden können in abgeleiteten Klassen überschrieben werden, um eigenen Quelltext einzubringen. In Abbildung 4.4 ist die vollständige Klassenstruktur dargestellt. Um die Generierung einer `case`-Methode zu veranschaulichen, wird ein kleines Beispiel betrachtet. In PL0 ist die folgende Grammatikregel Bestandteil der Spezifikation:

```
condition = ...
| {relop_condition} [left_side]:expression relop [right_side]:expression;
```

Für diese Regel werden folgenden Methoden generiert:

- `caseARelopConditionCondition` (Schnittstelle `Analysis`)
- `inARelopConditionCondition` (Klasse `DepthFirstAdapter`)
- `outRelopConditionCondition` (Klasse `DepthFirstAdapter`)

Die erzeugte Implementierung der `case`-Methode ist in Listing 4.5 dargestellt.

```
1  public void caseARelopConditionCondition(ARelopConditionCondition
       node)
2      {
3          inARelopConditionCondition(node);
4          if(node.getLeftSide() != null)
5          {
```

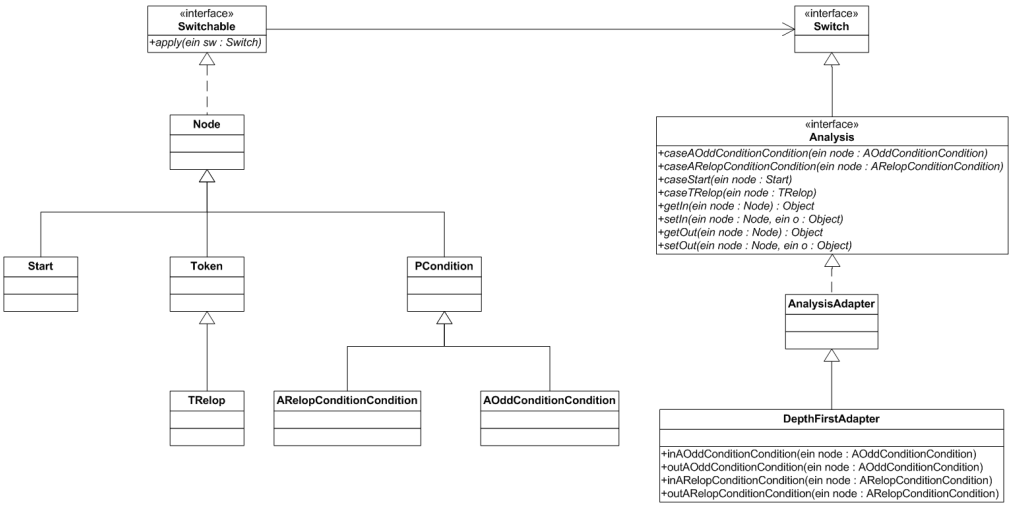


Abbildung 4.4: Vollständig von SableCC generierte Klassenstruktur

```
6               node.getLeftSide().apply(this);
7           }
8           if(node.getRelop() != null)
9           {
10              node.getRelop().apply(this);
11          }
12          if(node.getRightSide() != null)
13          {
14              node.getRightSide().apply(this);
15          }
16          outARelopConditionCondition(node);
17      }
```

Listing 4.5: Beispiel *caseARelopConditionCondition* -Methode

In den Zeilen 6, 10 und 14 wird zu den Subknoten abgestiegen. Zeile 3 und 16 sind dabei von besonderem Interesse. Hier werden die Methoden aufgerufen, in denen in den abgeleiteten Klassen der Nutzer eigenen Quelltext einbringen kann. Wie man daran sieht, werden die Möglichkeiten des Baumdurchlaufens damit sehr eingeschränkt (siehe nächsten Abschnitt).

4.1.4 Nachteile von SableCC

Im Rahmen der Implementierung des OCL2-Parsers [Kon] wurde festgestellt, dass SableCC ungeeignet für die Implementierung von OCL ist. Um einen konkreten OCL-Ausdruck in ein abstraktes OCL-Modell zu transformieren, ist es notwendig Attribute zu definieren und auszuwerten. Insbesondere soll es möglich sein, auf Attribute zugreifen zu können, die schon ausgewertet wurden (beispielsweise soll es möglich sein, zu prüfen, ob eine Variable schon definiert wurde). Doch dies ist mit SableCC nicht möglich, da den `in`- und `out`-Methoden keine Parameter übergeben werden können. Einen weiteren Nachteil betrifft den Bezug zur abstrakten OCL-Syntax. Dieser kann in der Spezifikation

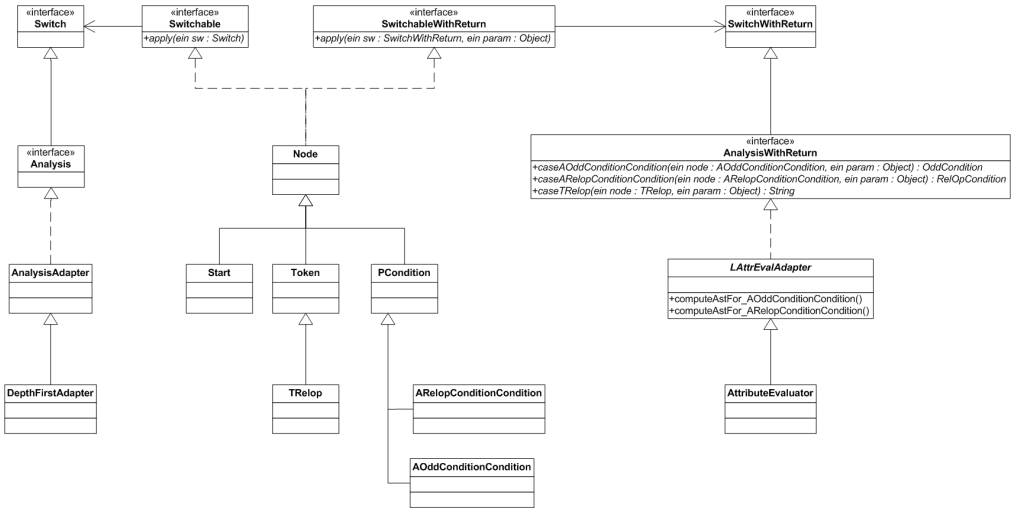


Abbildung 4.5: Besuchermuster erweitertes SableCC

nicht hergestellt werden. Um diese Nachteile auszugleichen, wurde im Rahmen von [Kon] SableCC erweitert. Diese Erweiterungen werden im nächsten Abschnitt vorgestellt.

4.2 Erweitertes SableCC

Um die im vorherigen Abschnitt dargestellten Nachteile zu umgehen, wurde im Rahmen von [Kon] der Compiler-Compiler SableCC angepasst, im weiteren Verlauf *erweitertes SableCC* genannt. Die Anpassung besteht aus drei Teilen: zum einen wurde das Besuchermuster der originalen Implementierung erweitert, es wurden neue Schlüsselwörter eingefügt, die die Quelltexterstellung beeinflussen, und es wurde ein Typsystem eingeführt, mit dessen Hilfe Bezug zur abstrakten Syntax hergestellt werden kann.

4.2.1 Erweitertes Besucher-Muster

In Abbildung 4.5 ist das Besuchermuster dargestellt, wie es im erweiterten SableCC verwendet wird.

Auf der linken Seite ist die ursprüngliche Besucherhierarchie und in der Mitte die originale Elementhierarchie dargestellt, wie sie SableCC generiert. Diese Teile werden in der Erweiterung ebenfalls generiert. Die Besucherhierarchie auf der rechten Seite der Abbildung kommt neu hinzu. Diese Hierarchie ähnelt der originalen Besucherhierarchie. Die Namen wurden ein wenig angepasst wie Tabelle 4.3 zeigt. Die Klasse `AttributeEvalutator` wurde nicht generiert. Sie wurde im Rahmen der PL0-Implementierung geschrieben und dient nur als Beispiel.

Die Signatur der `case`-Methoden innerhalb der Schnittstelle `AnalysisWithReturn` hat sich geändert. Jede dieser Methoden besitzt einen Rückgabewert (siehe Abschnitt 4.2.2) und einen Parameter `param` vom Typ `Object`. In Listing 4.6 ist beispielhaft eine Signatur dargestellt.

Elementtyp	alter Elementname	neuer Elementname
Schnittstelle	Analysis	AnalysisWithReturn
Klasse	AnalysisAdapter	LAttrEvalAdapter

Tabelle 4.3: Namensgebung im erweiterten SableCC

```
InputStatement caseAInputStatementStatement(AInputStatementStatement
    node, Object param) throws AttrEvalException;
```

Listing 4.6: Beispiel `case`-Methode

Der Typ des Parameters `param` wird etwas unglücklich generiert, denn nicht jedes Objekt kommt hier in Frage. Dieses Objekt muss Instanz der Klasse `Heritage` sein, die lediglich eine `copy`-Methode besitzt. Innerhalb der Implementierung wird `param` auf `Heritage` transformiert. Die Klasse `Heritage` dient dazu, während der Attributauswertung Werte aufzunehmen. Hier ist es beispielweise möglich eine Umgebung unterzubringen.

Damit der Parameter `param` von der Elementhierarchie weitergeben werden kann, implementiert die Klasse `Node` zusätzlich die Schnittstelle `SwitchableWithReturn`, die eine `apply`-Methoden mit dem zusätzlichen Parameter bereitstellt.

Die abstrakte Klasse `LAttrEvalAdapter` implementiert die Schnittstelle `AnalysisWithReturn`. Innerhalb der `case`-Methoden wird ein rekursiver Abstieg zu den Blättern des Baumes durchgeführt (von links nach rechts). Die konkret implementierte Klasse `DepthFirstAdapter` von SableCC besitzt `in`- und `out`-Methoden, die überschrieben werden können, um eigenen Quelltext einzubringen. In die Klasse `LAttrEvalAdapter` werden abstrakte Methoden mit dem Namen `compute` generiert, die in abgeleiteten Klassen implementiert werden müssen. An dieser Stelle kann eigener Quelltext eingefügt werden. In Abschnitt 4.2.3 wird näher auf die Implementierung eingegangen.

4.2.2 Typsystem

In dem erweiterten SableCC Compiler-Compiler wurde ein Typsystem hinzugefügt. Damit ist es nun möglich, Bezug auf die abstrakte Syntax einer Sprache zu nehmen. Typinformationen können sowohl an den Regeln als auch an den Morphemen gesetzt werden. Sie beeinflussen die Erzeugung der `case`-Methoden in der Schnittstelle `AnalysisWithReturn`. Das Typsystem soll an zwei Beispielen erläutert werden. Zunächst wird die Regeldefinition der Bedingung betrachtet (siehe Listing 4.7).

```
1 condition<Condition> =
2       {odd_condition}
3       <OddCondition> odd additive_expression
4       | {relop_condition}
5       <RelOpCondition> [left_side]:additive_expression
6       relop [right_side]:additive_expression;
```

Listing 4.7: Beispiel Condition-Regel mit Typen

Die Typinformation steht hinter dem Regelnamen und den Alternativnamen zwischen den Zeichen <und >. Im Beispiel findet man die Typinformationen hinter dem Regelnamen `condition` (Zeile 1) und am Anfang der Zeilen 3 und 5. Die Typen in diesem

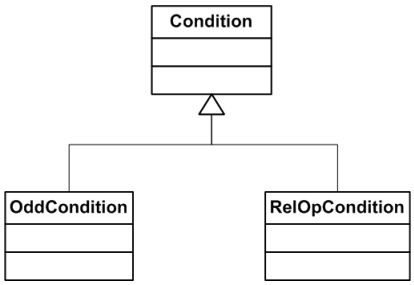


Abbildung 4.6: Klassenbeziehung der `Condition`-Typen

Beispiel beziehen sich auf die Klassen `Condition`, `OddCondition` und `RelOpCondition` aus der abstrakten Syntax von PL0 (siehe Abschnitt 3.4 auf Seite 50). Die Vererbungsbeziehung zwischen den Typen in diesem Beispiel ist in Abbildung 4.6 dargestellt.

Der Rückgabewert der `case`-Methode entspricht dem Typen der entsprechenden Alternativdefinition. Im Beispiel der Bedingung werden die Methoden wie in Listing 4.8 gezeigt, generiert.

```
1 OddCondition
2   caseAOddConditionCondition
3   (AOddConditionCondition node, Object param)
4     throws AttrEvalException;
5
6 RelOpCondition
7   caseARelopConditionCondition
8   (ARelopConditionCondition node, Object param)
9     throws AttrEvalException;
```

Listing 4.8: Beispiel generierte Methodensignaturen der Condition-Regel

Die Typinformationen finden sich hier als Rückgabewerte wieder (vgl. Zeile 1 und 6). Die Aufgabe des Parameters `param` wurde bereits erläutert (siehe Abschnitt 4.2.1 auf Seite 69). Der zweite Parameter referenziert das Element des konkreten Syntaxbaumes, das in der `case`-Methode bearbeitet wird (vgl. Besuchermuster).

Allgemein lässt sich sagen, dass die Typen auf der rechten Seite einer Regel Unterklassen des Typen auf der linken Seite (bzw. derselbe Type) der Regel sein müssen.

Auch kann eine Typinformation bei der Definition eines Morphems angegeben werden. In Listing 4.9 ist die Definition der Zahl dargestellt, die den Typ `Number` der abstrakten Syntax zugewiesen bekommt.

```
number<Number> = ['1'..'9']['0'..'9']* | '0';
```

Listing 4.9: Beispiel Morphemdefinition `number` mit Typangabe

Die Definition der `case`-Methode ist in Listing 4.10 dargestellt.

```
Number caseTNumber(TNumber node, Object param) throws
    AttrEvalException;
```

Listing 4.10: `case`-Methode des Nummermorphems

Die `case`-Methode besitzt in diesem Fall den gleichen Rückgabetyp wie die Morphemdefinition.

4.2.3 Struktur der `case`-Methoden

Die Klasse `LAttrEvalAdapter` implementiert die `case`-Methoden der Schnittstelle `AnalysisWithReturn`. Die Implementierung orientiert sich dabei an den Grammatikregeln. Für jedes Grammatiksymbol auf der rechten Seite einer Regel wird ein Abstieg zu diesem Symbol durchgeführt. Ein solcher Abstieg liefert eine Instanz der abstrakten Syntax zurück. Nachdem alle Symbole einer Regel abgearbeitet wurden, wird eine Instanz der abstrakter Syntax mit Hilfe einer Fabrikklasse erzeugt, die dem Rückgabetyp der aktuellen Alternativen entspricht. Anschließend wird eine `compute`-Methode aufgerufen. Dieser Methode wird die gerade eben erzeugte Instanz der Fabrikklasse übergeben, das aktuelle `Heritage`-Objekt und die zuvor berechneten Instanzen der abstrakten Syntax. Diese `compute`-Methode ist abstrakt und muss in Subklassen überschrieben werden. An dieser Stelle kann eigener Quelltext eingefügt werden.

Für die relationale Bedingung aus Listing 4.8 wird die in Listing 4.11 gezeigte `case`-Methode generiert.

```
1  public final RelOpCondition
2    caseARelopConditionCondition
3    (ARelopConditionCondition node, Object param)
4    throws AttrEvalException {
5
6    Heritage nodeHrtg = (Heritage) param;
7          Heritage childHrtg = null;
8
9          PAdditiveExpression childLeftSide = node.getLeftSide();
10         Expression astLeftSide = null;
11         if( childLeftSide != null) {
12             ...
13             astLeftSide = (Expression) childLeftSide.apply(this,
                   nodeHrtg.copy());
14             ...
15         }
16         TRelop childRelop = node.getRelop();
17         String astRelop = null;
18         if( childRelop != null) {
19             ...
20             astRelop = (String) childRelop.apply(this, nodeHrtg.copy
                   ());
21             ...
22         }
23         PAdditiveExpression childRightSide = node.getRightSide();
24         Expression astRightSide = null;
25         if( childRightSide != null) {
26             ...
27             astRightSide = (Expression) childRightSide.apply(this,
                   nodeHrtg.copy());
28             ...
29         }
30         RelOpCondition myAst = (RelOpCondition) factory.createNode("
               RelOpCondition");
31         myAst = computeAstFor_ARelopConditionCondition(myAst,
               nodeHrtg,
32              astLeftSide,
33              astRelop,
```

```
34              astRightSide);
35          return myAst;
36      }
```

Listing 4.11: Beispiel generierte `case`-Methode für die Alternative ARelopConditionCondition

In den Zeilen 13, 20 und 28 finden die Kindaufrufe statt. In Zeile 31 wird mittels einer Fabrikklasse ein Element aus der abstrakten Syntax erzeugt und in Zeile 32 wird die `compute`-Methode aufgerufen.

Im nächsten Abschnitt werden die syntaktischen Elemente erläutert, die neu hinzugekommen sind und die die Generierung der `case`-Methoden beeinflussen.

4.2.4 Syntaktische Elemente

SableCC wurde um neue Schlüsselwörter erweitert: `#chain`, `#maketree`, `#customheritage`, `#nocreate` und `!`. Die ersten vier Schlüsselwörter können in einer Regeldefinition verwendet werden, währenddessen das Ausrufezeichen ! bei der Definition von Morphemen gesetzt werden kann. Alle Schlüsselwörter steuern die Generierung der `case`-Methoden in der Klasse `LAttrEvalAdapter`.

Die Markierung !

Das Ausrufezeichen ! kann nur vor Morphemen gesetzt werden, wie in Listing 4.12 dargestellt. Es ist zu erkennen, dass das Ausrufezeichen ! ausschließlich vor Schlüsselwörtern gesetzt wurde. Es bewirkt, dass der Knoten im Baum übergangen wird.

```
Tokens
  relop = ('<=' | '>=' | '<' | '>' | '#');
  aop = ('+' | '-');
  mop = ('*' | '/');
  !if = 'IF';
  !while = 'WHILE';
  !begin = 'BEGIN';
  !end = 'END';
  !procedure = 'PROCEDURE';
  !call = 'CALL';
  !do = 'DO';
  !then = 'THEN';
  !const = 'CONST';
  !var = 'VAR';
  !odd = 'ODD';
  !exclam = '!';
  !quest = '?';
  !equal = '=';
  !semicolon = ';';
  !comma = ',';
  !point = '.';
  !assign = ':=';
  !openparen = '(';
  !closeparen = ')';
  number<Number> = ['1'..'9']['0'..'9']* | '0';
  ident<Nameable> = name;
  blank = ( ' ' | 10 | 13 | 9 )*;
  operatortype = '+' | '-' | '*' | '/';
```

Listing 4.12: Ausrufezeichen ! in der Morphemdefinition

Für die If-Bedingung gibt es in der Spezifikation folgende Grammatikregel:

```
statement<Statement> = ...
| {if_statement}<IfStatement> if condition then statement
```

In Listing 4.13 ist die dazugehörige `case`-Methode dargestellt. Die If-Anweisung enthält die Schlüsselwörter `IF` und `THEN`. Ein Abstieg zu diesen Kindelementen ist nicht erforderlich, da sie keine relevanten Informationen tragen. Da die Schlüsselwörter in der abstrakten Syntax keine Rolle spielen, ist dies eine nützliche Abkürzung.

```
1  public final IfStatement caseAIfStatementStatement(
2    AIfStatementStatement node,
3    Object param)
4    throws AttrEvalException {
5          Heritage nodeHrtg = (Heritage) param;
6          Heritage childHrtg = null;
7
8          PCondition childCondition = node.getCondition();
9          Condition astCondition = null;
10         if( childCondition != null) {
11             ...
12             astCondition = (Condition)
13       childCondition.apply(this, nodeHrtg.copy());
14             ...
```

```
15          }
16          PStatement childStatement = node.getStatement();
17          Statement astStatement = null;
18          if( childStatement != null) {
19              ...
20              astStatement = (Statement)
21      childStatement.apply(this, nodeHrtg.copy());
22              ...
23          }
24          IfStatement myAst = (IfStatement)
25      factory.createNode("IfStatement");
26          myAst = computeAstFor_AIfStatementStatement(
27      myAst,
28      nodeHrtg,
29                  astCondition,
30                  astStatement);
31          return myAst;
32      }
```

Listing 4.13: Beispiel `case`-Methode der If-Anweisung

Das Schlüsselwort #chain

`#chain` kann immer in Regeln eingesetzt werden, die nur ein Nichtterminal auf der rechten Seite haben (mehrere Alternativen sind erlaubt). Durch `#chain` wird keine `compute`-Methode generiert, sondern das Ergebnis des einzigen Kindknotens (das ist der Grund, warum nur ein Nichtterminal auf der rechten Seite stehen darf) wird als Gesamtergebnis zurückgegeben. Als Beispiel dient die Regel für die Befehlsaufzählung (`statementenmum`) (Listing 4.14).

```
statementenum<Statement> = semicolon statement #chain;
```

Listing 4.14: #chain in der Regeldefinition *statementenum*

Die entsprechende `case`-Methode ist in Listing 4.15 dargestellt.

```
1     public final Statement caseAStatementenum(AStatementenum node,
          Object param) throws AttrEvalException {
2         Heritage nodeHrtg = (Heritage) param;
3         Heritage childHrtg = null;
4
5         PStatement childStatement = node.getStatement();
6         Statement astStatement = null;
7         if( childStatement != null) {
8             ...
9             astStatement = (Statement) childStatement.apply(this,
                  nodeHrtg.copy());
10            ...
11        }
12        Statement myAst = astStatement;
13        return myAst;
14    }
```

Listing 4.15: `case`-Methode ohne `compute`-Methode

Hier wird keine `compute`-Methode aufgerufen (Zeile 12). Statt dessen wird das Ergebnis der Kindberechnung einfach weitergegeben. An dieser Stelle sei anzumerken, dass die Typen natürlich konform sein müssen. Eine Besonderheit muss berücksichtigt werden: auf der rechten Seite einer Regeldefinition können sich mehrere Grammatiksymbole befinden, sofern nur genau eines von diesen berücksichtigt wird. Die anderen Grammatiksymbole müssen mit dem Ausrufezeichen ! gekennzeichnet sein. Unter diesen Umständen ist die Verwendung des Schlüsselwortes `#chain` trotzdem möglich. In Listing 4.16 ist dafür ein Beispiel dargestellt.

```
1  unary_expression<Expression> = ...
2    | {expression}<Expression> openparen additive_expression
       closeparen#chain;
```

Listing 4.16: Benutzung des Schlüsselwortes textit#chain trotz mehrerer Grammatiksymbole

`openparen` und `closeparen` sind zwei Morpheme und werden ignoriert (Markierung ! in der Morphemdefinition). Daher kann `#chain` an dieser Stelle ebenfalls eingesetzt werden.

Das Schlüsselwort #nococreate

Das Schlüsselwort #nococreate verhindert, dass ein Element der abstrakten Syntax mittels der Fabrikklasse gebaut wird. Folglich wird es nicht der `compute`-Methode übergeben. Als Beispiel soll die Definition der Ausgabeanweisung dienen (siehe Listing 4.17).

```
statement<Statement> =
  ...
  | {output_statement}<OutputStatement>
  exclam additive_expression #nocreate
  ...
```

Listing 4.17: OutputStatement mit #nocreate

Die entsprechende `case`-Methode ist in Listing 4.18 dargestellt.

```
1  public final OutputStatement caseAOutputStatementStatement(
     AOutputStatementStatement node, Object param) throws
     AttrEvalException {
2          Heritage nodeHrtg = (Heritage) param;
3          Heritage childHrtg = null;
4
5          PAdditiveExpression childAdditiveExpression = node.
               getAdditiveExpression();
6          Expression astAdditiveExpression = null;
7          if( childAdditiveExpression != null) {
8              ...
9              astAdditiveExpression = (Expression)
                   childAdditiveExpression.apply(this, nodeHrtg.copy())
                   ;
10             ...
11         }
12         OutputStatement myAst =
               computeAstFor_AOutputStatementStatement(nodeHrtg,
13             astAdditiveExpression);
```

```
14          return myAst;
15      }
```

Listing 4.18: `case`-Methode des OutputStatement

Der Knoten des abstrakten Modells muss in diesem Fall selbst erzeugt werden. In Zeile 12 wird kein generiertes Objekt übergeben. Das kann nützlich sein, wenn die Fabrikklassen das Erzeugen des betroffenen Knotens nicht unterstützt.

Das Schlüsselwort #customheritage

Das Schlüsselwort `#customheritage` kann hinter einem Grammatiksymbol, das nicht durch das Ausrufezeichen in der Morphemdefinition gekennzeichnet ist, geschrieben werden. Es bewirkt, dass eine neue Methode in die Klasse `LAttrEvalAdapter` generiert wird, die den Präfix `inside` trägt. Der weitere Name setzt sich aus dem Namen der generierten Klasse für diese Alternative, dem Infix `_computeHeritageFor_` und dem Namen des Grammatiksymbols, nach dem das Schlüsselwort gesetzt wurde, als Postfix zusammen. Das Schlüsselwort wird beispielsweise in der folgenden Grammatikregel verwendet:

```
vardecl<List> = var ident #customheritage varenum* semicolon;
```

Daraus ergibt sich der Methodenname

```
insideAVardecl_computeHeritageFor_Ident
```

Das Schlüsselwort bewirkt außerdem, dass die Traversierung des Baumes an der Stelle, an der es gesetzt wurde, kurzzeitig unterbrochen wird, und Code an dieser Stelle eingefügt werden kann. Die `case`-Methode wird daraufhin angepasst. Für das oben eingeführte Beispiel ergibt sich der in Listing 4.19 aufgezeigte Quelltext.

```
1  public final List caseAVardecl(AVardecl node, Object param) throws
       AttrEvalException {
2          Heritage nodeHrtg = (Heritage) param;
3          Heritage childHrtg = null;
4
5          TIdent childIdent = node.getIdent();
6          Nameable astIdent = null;
7          if( childIdent != null) {
8              childHrtg = insideAVardecl_computeHeritageFor_Ident(node
                   , childIdent, nodeHrtg.copy());
9              if ( childHrtg == null ) {
10                 childHrtg = nodeHrtg.copy();
11             }
12             ...
13             astIdent = (Nameable) childIdent.apply(this, childHrtg )
                   ;
14             ...
15         }
16         List astListVarenum = new java.util.LinkedList();
17         {
18             PVarenum childVarenum = null;
19             Variable astVarenum = null;
20             Object temp[] = node.getVarenum().toArray();
```

```
21              for(int i = 0; i < temp.length; i++) {
22                  childVarenum = (PVarenum) temp[i];
23                  ...
24                  astVarenum = (Variable) childVarenum.apply(this,
                        nodeHrtg.copy());
25                  ...
26                  astListVarenum.add(astVarenum);
27              }
28          }
29          List myAst = (List) factory.createNode("List");
30          myAst = computeAstFor_AVardecl(myAst, nodeHrtg,
31              astIdent,
32              astListVarenum);
33          return myAst;
34      }
```

Listing 4.19: `case`-Methode der Variablendeklaration

Wie in Zeile 8 zu sehen ist, wird die `inside`-Methode aufgerufen, bevor in den Zweig des Identifizierers abgestiegen wird (Zeile 13). In dieser Methode kann das `Heritage`-Objekt geändert werden, was sich dann auf die Berechnung des Identifizierers auswirkt. In diesem Fall hält das `Heritage`-Objekt eine Instanz der Klasse `Environment`, die Variablen, Prozeduren und Konstanten aufsammelt (siehe Abschnitt 3.5.1 auf Seite 56). In diese Umgebung wird der aktuelle Identifizierer als Variable abgespeichert, sofern er noch nicht als Konstante oder Prozedur existiert. Das `Heritage`-Objekt wird in Zeile 8 kopiert, nicht aber die `Environment`-Instanz innerhalb, so dass sich die Änderungen der `Environment`-Instanz auch über die Berechnung des Identifizierers (Zeile 13) hinaus wirkt. So kann im weiteren Verlauf auf die Variable zugegriffen werden.

Die eigentliche Eintragung des Identifizierers bzw. der Variable in die Umgebung erfolgt in der Methode `createNodeFor_TIdent`, die für das Morphem `ident` erzeugt wird. Wenn die `compute`-Methode für den Variablennamen (`ident`) innerhalb der `caseAVardecl`-Methode aufgerufen wird, dann weiß sie nicht, ob es sich bei diesem Bezeichner um einen Variablennamen, Konstantennamen oder einen Prozedurnamen handelt. Diese Information ist aber durch die `compute`-Methode der jeweiligen Deklaration gegeben. Um diese Information an die Bezeichnermethode weiterzureichen, wird sich hier dem `Heritage`-Objekt bedient. Die `Heritage`-Klasse besitzt ein Attribut namens `isVariableDeclaration`, das auf *true* gesetzt wird, wenn die `Heritage`-Instanz von einer Variablendeklaration abgesetzt wird. Um die Heritage-Instanz manipulieren zu können, wird sich hier der `inside`-Methode bedient. In dieser wird das passende Attribut auf *true* gesetzt. Innerhalb der `createNodeFor_TIdent`-Methode wird auf dieses Attribut geprüft. So kann entschieden werden, welche semantischen Prüfungen durchzuführen sind. In diesem Fall wird in der Umgebung der `Heritage`-Instanz nachgesehen, ob es bereits eine Konstante oder eine Prozedur mit dem gleichen Namen gibt. Wenn ja, wird eine Fehlermeldung ausgegeben. Andernfalls wird eine neue Variable erzeugt, in die Umgebung hinzugefügt und zurückgegeben.

Das Schlüsselwort #maketree

Mit `#maketree` ist es möglich eine Liste von Knoten des konkreten Syntaxbaumes in einen Baum der abstrakten Syntax zu überführen. Zur Veranschaulichung soll folgendes Beispiel dienen: Arithmetische Ausdrücke kann man mit Hilfe der Grammatik in Abbildung 4.7 beschreiben.

AdditiveExpression	→	MultiplicativeExpression
AdditiveExpression	→	MultiplicativeExpression AdditiveExpressionTail+
AdditiveExpressionTail	→	additiveOperator MultiplicativeExpression
MultiplicativeExpression	→	UnaryExpression
MultiplicativeExpression	→	UnaryExpression MultiplicativeExpressionTail+
MultiplicativeExpressionTail	→	mulitplicativeOperator UnaryExpession
UnaryExpression	→	'('additiveOperator AdditiveExpression')'
UnaryExpression	→	ident
UnaryExpression	→	number
UnaryExpression	→	'(' AdditiveExpression ')'
additiveOperator	→	+ | -
multiplicativeOperator	→	* | \

Abbildung 4.7: Grammatik der arithmetischen Ausdrücke

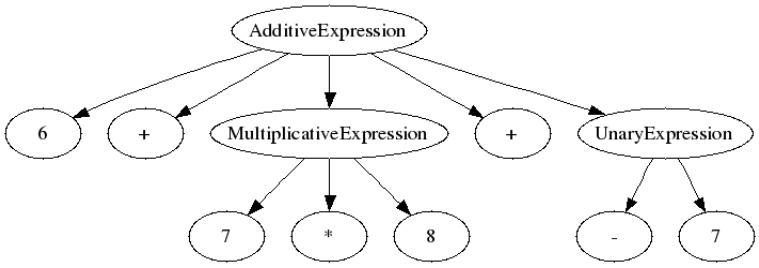


Abbildung 4.8: Listenrepräsentation des arithmetischen Ausdrucks 6+7*8+(-7)

Die Metasymbole entsprechen denen aus Tabelle 4.1.1 (Abschnitt 4.1.1 auf Seite 62). Die arithmetischen Ausdrücke bestehen in dieser Grammatik immer aus einem Kopf und einer Liste von *Tail*-Elementen, von dem jedes den Operator und den rechten Operanden mit sich führt. Diese Listen kann man nun mittels `#maketree` in einen Baum umwandeln. Dazu wird folgender arithmetischer Ausdruck betrachtet: 6+7*8+(-7). Dieser Ausdruck wird in seiner Listendarstellung in Abbildung 4.8 dargestellt.

Mittels der `#maketree`-Anweisung kann die Listendarstellung in eine Baumdarstellung transformiert werden (siehe Abbildung 4.9).

In der Beispielgrammatik PL0 wurden die arithmetischen Ausdrücken nach der zuvor vorgestellten Grammatik (Abbildung 4.7) implementiert. Die Zeilen der SableCC-Spezifikation sind in Listing 4.20 dargestellt.

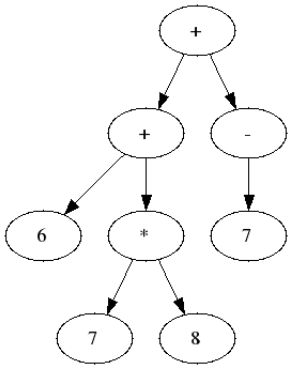


Abbildung 4.9: Baumrepräsentation des arithmetischen Ausdrucks 6+7*8+(-7)

```
additive_expression<Expression> =
{chain}<Expression> multiplicative_expression #chain

| {binary}<BinaryOperator> multiplicative_expression
    additive_expression_tail+#maketree;


additive_expression_tail<BinaryOperator> = aop
    multiplicative_expression;


multiplicative_expression<Expression> =
{chain} <Expression> unary_expression #chain

| {binary}<BinaryOperator> unary_expression
    multiplicative_expression_tail+#maketree;


multiplicative_expression_tail<BinaryOperator> = mop
    unary_expression;


unary_expression<Expression> =
{aop}<UnaryOperator> openparen aop additive_expression closeparen
| {ident}<Value> ident#customheritage#nocreate
| {number}<Number> number#chain
| {expression}<Expression> openparen additive_expression closeparen#
    chain;
```

Listing 4.20: arithmetische Ausdrücke in PL0

Wie in der SableCC-Spezifikation zu sehen ist, steht das Schlüsselwort `#maketree` hinter den *Tail*-Nichtterminalen. Es hat zur Wirkung, dass eine

`insideXXX_computeHeritageFor_XXX`

Methode generiert wird (Namensbildung analog zum Schlüsselwort `#customheritage`).

Neu ist hier, dass sich die Parameter geändert haben. Von dieser Methode wird ebenso von der generierten Methode über das Schlüsselwort `#customheritage` verlangt, ein `Heritage`-Objekt zurückzugeben. Die `inside`-Methode bekommt alle zuvor berechneten Werte und auch die Knoten aus dem konkreten Syntaxbaum. Die *Tail*-Elemente sind jeweils Listen. So bekommt die `inside`-Methode ebenfalls das vorhergehende Element übergeben. Wenn das erste *Tail*-Element berechnet wird, ist dieses Element `null`. Erst das zweite *Tail*-Element hat einen Vorgänger. Dieses Attribut besitzt den Namen `astPreviousSilbing`. In Listing 4.21 ist die Signatur der `inside`-Methode des additiven Ausdrucks dargestellt.

```
1  public Heritage
2  insideABinaryAdditiveExpression_computeHeritageFor_AdditiveExpressionTail
       (
3    ABinaryAdditiveExpression parent,
4    PAdditiveExpressionTail child,
5    Heritage parentHrtgCopy,
6    BinaryOperator astPreviousSibling,
7    Expression astMultiplicativeExpression
8  )
```

Listing 4.21: Signatur der `inside`-Methode für additiven Ausdruck

Die zweite Zutat, um einen Baum eines arithmetischen Ausdruckes aufzubauen, ist wieder die `Heritage`-Instanz. Diese speichert ein Attribut namens `source` vom Typ `Expression`. Dieses Attribut wird in der jeweiligen `inside`-Methode gesetzt. Wie der Name `source` andeutet, wird diese `Expression` als Quelle für die weiteren Berechnungen verwendet. `#maketree` passt aber auch die zugehörige `case`-Methode an. In Listing 4.22 ist die `case`-Methode dargestellt.

```
1  public final BinaryOperator caseABinaryAdditiveExpression(
2    ABinaryAdditiveExpression node,
3    Object param)
4    throws AttrEvalException {
5          ...
6  PMultiplicativeExpression childMultiplicativeExpression =
7    node.getMultiplicativeExpression();
8  Expression astMultiplicativeExpression = null;
9  if( childMultiplicativeExpression != null) {
10   ...
11   astMultiplicativeExpression =
12   (Expression) childMultiplicativeExpression.apply(this, nodeHrtg.
         copy());
13   ...
14 }
15
16 BinaryOperator astTreeAdditiveExpressionTail = null;
17 {
18   PAdditiveExpressionTail childAdditiveExpressionTail = null;
19   BinaryOperator astPreviousAdditiveExpressionTail = null;
20   Object temp[] =
21   node.getAdditiveExpressionTail().toArray();
22   for(int i = 0; i < temp.length; i++) {
23     childAdditiveExpressionTail =
24     (PAdditiveExpressionTail) temp[i];
```

```
25      childHrtg =
26      insideABinaryAdditiveExpression_computeHeritageFor_AdditiveExpressionTail
        (
27        node,
28        childAdditiveExpressionTail,
29        nodeHrtg.copy(),
30        astPreviousAdditiveExpressionTail,
31        astMultiplicativeExpression
32      );
33      if ( childHrtg == null ) {
34        childHrtg = nodeHrtg.copy();
35      }
36      ...
37      astTreeAdditiveExpressionTail = (BinaryOperator)
38        childAdditiveExpressionTail.apply(this, childHrtg);
39      ...
40      astPreviousAdditiveExpressionTail =
            astTreeAdditiveExpressionTail;
41      }
42    }
43  BinaryOperator myAst = (BinaryOperator)
44    factory.createNode("BinaryOperator");
45  myAst = computeAstFor_ABinaryAdditiveExpression(
46    myAst,
47    nodeHrtg,
48    astMultiplicativeExpression,
49    astTreeAdditiveExpressionTail
50    );
51  return myAst;
52  }
```

Listing 4.22: `case`-Methode für additiven Ausdruck

Die wichtigen Zeilen sind Zeile 26, 38 und 41. In Zeile 26 wird die `inside`-Methode aufgerufen, in Zeile 38 wird der Abstieg zum *Tail*-Element durchgeführt und in Zeile 41 wird der Vorgängerknoten neu belegt; mit dem Wert, den der Abstieg in das *Tail*-Element gerade berechnet hat. Dieses wird der Berechnung des nächstes *Tail*-Elementes übergeben. Erwähnenswert ist auch, dass der abschließenden `compute`-Methode keine Liste von *Tail*-Elementen übergeben wird, sondern ausschließlich das letzte berechnete Element. Es ist offensichtlich, dass an diesem Element der aufgebaute Baum hängen muss.

Um die Funktionsweise zu erklären, wird der folgende arithmetische Ausdruck betrachtet: 3+5+7. Die `case`-Methode des additiven arithmetischen Ausdrucks wird aufgerufen und das erste Element (hier die 3) in Zeile 11 berechnet. Nun wird eine Iteration für das erste *Tail*-Element angestoßen. Die `inside`-Methode bekommt den vorher berechneten Ausdruck und als `astPreviousSilbing null` übergeben (da bisher kein *Tail*-Element berechnet wurde). Innerhalb der `inside`-Methode muss nun entschieden werden, wie das `source`-Attribut der Heritage-Instanz gesetzt werden muss. In diesem Fall muss es der zuvor berechnete Ausdruck sein und nicht `astPreviousSilbing`. Dieses `source`-Attribut wird dann als linker Operand verwendet. Die `inside`-Methode für die additiven arithmetischen Ausdrücke ist in Listing 4.23 dargestellt.

```
1  public Heritage
       insideABinaryAdditiveExpression_computeHeritageFor_AdditiveExpressionTail
       (
2    ABinaryAdditiveExpression parent, PAdditiveExpressionTail child,
3    Heritage parentHrtgCopy, BinaryOperator astPreviousSibling,
4    Expression astMultiplicativeExpression) throws AttrEvalException {
5
6    if (astPreviousSibling != null) {
7      parentHrtgCopy.setSource(astPreviousSibling);
8    } else if (astMultiplicativeExpression != null) {
9      parentHrtgCopy.setSource(astMultiplicativeExpression);
10   } else {
11     throw new RuntimeException("Internal error");
12   }
13   return parentHrtgCopy;
14 }
```

Listing 4.23: `inside`-Methode für additiven Ausdruck

Nachdem die Heritage-Instanz verändert wurde, wird in das erste *Tail*-Element hinabgegangen und berechnet. Dabei wird die Methode

`computeAstFor_AAdditiveExpressionTail`

aufgerufen. Diese Methode bekommt als Parameter die `Heritage`-Instanz, eine Zeichenkette, die entweder + oder - enthält, und eine Instanz vom Typ `Expression` übergeben. Zur Erinnerung, die entsprechende Zeile in der SableCC-Spezifikation lautet

```
additive_expression_tail<BinaryOperator> = aop multiplicative_expression;
```

Wie in dieser Zeile schon angedeutet ist, muss diese `compute`-Methode eine Instanz vom Typ `BinaryExpression` zurückgeben. Bisher wird ihr aber nur der Operator und der rechte Operand übergeben. Den linken Operanden erhält sie über die Heritage-Instanz. Die gesamte `compute`-Methode ist in Listing 4.24 dargestellt.

```
1  public BinaryOperator computeAstFor_AAdditiveExpressionTail(
2    BinaryOperator myAst,
3    Heritage nodeHrtg,
4    String astAop,
5    Expression astMultiplicativeExpression) throws AttrEvalException {
6
7    OperatorType opType = tokenType2OperatorType(astAop);
8
9    BinaryOperator result = new BinaryOperator();
10   result.setRightOperand(astMultiplicativeExpression);
11   result.setOpType(opType);
12   result.setLeftOperand(nodeHrtg.getSource());
13   return result;
14   }
```

Listing 4.24: `compute`-Methode für additiven Tail-Ausdruck

Wie in der `compute`-Methode zu sehen ist, stammt der linke Operand tatsächlich aus der `Heritage`-Instanz. Nun wurde der arithmetische Ausdruck 3+5 ausgewertet und es verbleibt noch das *Tail*-Element +7. Die `case`-Methode innerhalb des additiven

Ausdrucks (`caseABinaryAdditiveExpression`) weist dem Vorgänger den gerade eben berechneten Teilbaum zu

```
astPreviousAdditiveExpressionTail = astTreeAdditiveExpressionTail
```

Dieser wird wieder in die *inside*-Methode gesteckt. Da es nun einen Vorgänger gibt, wird dieser verwendet. Es wird wieder die `compute`-Methode des *Tail*-Elementes aufgerufen und der linke Operand (aus der `Heritage`-Instanz), der Operator und der rechte Operand gesetzt. Somit ist die Berechnung des Ausdruckes fast vollständig. Es folgt noch die Ausführung der `computeAstFor_ABinaryAdditiveExpression`-Methode (siehe Listing 4.25).

```
1 public BinaryOperator computeAstFor_ABinaryAdditiveExpression(
2   BinaryOperator myAst, Heritage nodeHrtg,
3   Expression astMultiplicativeExpression,
4   BinaryOperator astTreeAdditiveExpressionTail)
5   throws AttrEvalException {
6
7   return astTreeAdditiveExpressionTail;
8 }
```

Listing 4.25: `compute`-Methode für additiven Gesamtausdruck

Innerhalb dieser Methode muss lediglich der Teilbaum zurückgegeben werden, der sich hinter dem Parameter `astTreeAdditiveExpressionTail` verbirgt. Der Parameter `astMultiplicativeExpression` kann ignoriert werden, da dieser Ausdruck schon im Baum vorhanden ist.

4.3 Verwendung des erweiterten SableCC

An dieser Stelle erfolgt eine kurze Zusammenfassung für das Arbeiten mit der SableCC-Erweiterung. Ausführlicher ist dies in Kapitel 6.2 auf Seite 101 dargestellt.

Folgende Schritte sind erforderlich:

1. Schreiben der SableCC-Spezifikation der Morpheme und der Grammatik für eine Sprache
2. Aufstellen der abstrakten Syntax (implementieren einer Klassenhierarchie)
3. Grammatikspezifikation mit Typen anreichern
4. die verwendeten Typen in die Klasse `org.sablecc.sablecc.TypeMap` eintragen
5. `import`-Anweisungen in die Datei `analyses.txt` einfügen
6. Klasse `Heritage` implementieren und erreichbar machen (siehe Punkt 5)
7. Klasse `NodeFactory` implementieren und erreichbar machen (siehe Punkt 5)
8. Generierung eines Parsers für die Sprache
9. Implementieren der Transformation des konkreten Syntaxbaumes in einen abstrakten

Punkt 4 und 5 sind notwendig, da diese Informationen nicht in der Grammatikspezifikation eingetragen werden können. Dies könnte in der SableCC-Implementierung geändert werden. Die Punkte 6 und 7 könnten ebenfalls automatisiert werden, indem eine Standardimplementierung vorgenommen wird, die in ein von den generierten Verzeichnissen kopiert wird.

4.4 Zusammenfassung

Der Parser-Generator SableCC ist eine echte Alternativen zu den Generatoren, deren Spezifikationsdateien mit Aktionen angereichert werden können. Diese Anreicherung führt dazu, dass die Spezifikation unübersichtlicher wird und der Nutzer sich nicht vollständig auf die Grammatik- und Morphemdefinition konzentrieren kann. Sehr gut ist vor allem die Unterstützung von LALR(1)-Grammatiken, da der Nutzer so einen großen Teil an Sprachen abdecken kann und Grammatiken einfacher zu formulieren sind (Linksrekursion ist zugelassen).

Fragwürdig erscheint das erweiterte Besuchermuster. Ziel dieser Erweiterung ist es, die Elementhierarchie des Besuchermusters flexibel zu gestalten. Da das Framework bei jeder Änderung der Spezifikation neu generiert wird, erscheint der Aufwand dieser Änderung nicht gerechtfertigt.

In [Kon] wurde SableCC um viele Möglichkeiten erweitert. Die vorteilshafte Änderung ist das Einfügen des Typsystems, da es so möglich ist, auf eine abstrakte Syntax zugreifen zu können. Das Schlüsselwort `#chain` und die Markierung `!` erleichtern die Implementierung der Attributauswertung teilweise erheblich. Das Schlüsselwort `#customheritage` bietet eine Möglichkeit an, eine Umgebung in die Berechnung einzubringen. Allerdings muss manchmal sehr genau überlegt werden, wie diese Auswertung erfolgen soll, da die generierten Klassen nicht mit einer Umgebung umgehen, sondern mit einer Instanz der Klasse `Heritage`. Das Schlüsselwort `#maketree` ist sehr kompliziert. Der Nutzer muss sich im Klaren darüber sein, wie der generierten Quelltext arbeitet.

Ohne diese Erweiterung wäre die Implementierung des OCL-Parsers (siehe Kapitel 6 auf Seite 99) nicht ohne erheblichen Aufwand möglich gewesen, da viele Parser-Generatoren nicht die Sprachklasse LALR(1) und ein Typsystem unterstützen.

5 JastAdd

In diesem Kapitel soll der Attributauswerter *JastAdd* [JASTWeb] vorgestellt werden, da er für die Implementierung des OCL-Parsers eine wichtige Rolle spielt. Zunächst wird ein kurzer Überblick über JastAdd im Allgemeinen und der Architektur gegeben. Um ein Gefühl für JastAdd zu bekommen, wurde die kleine Sprache PL0 (siehe Abschnitt 3, Seite 43) mit JastAdd implementiert. Diese Implementierung wird im zweiten Teil dieses Kapitels vorgestellt.

5.1 Einführung in JastAdd

In Kapitel (siehe Abschnitt 2.2, Seite 22) wurde beschrieben, wie ein Parser aus einem Quelltext einen konkreten Syntaxbaum erzeugt. Darüber hinaus wurde im Kapitel (siehe Abschnitt 2.4 auf Seite 34) die abstrakte Syntax vorgestellt, deren Instanzen abstrakte Syntaxbäume oder abstrakte Modelle sind. Den abstrakten Syntaxbaum möchte man oft in einen abstrakten Syntaxgraphen transformieren, indem beispielsweise Variablenzuweisungen nicht in Form von Zeichenketten gespeichert werden, sondern als Objektreferenzen. Auch sollen während dieser Transformation semantischen Prüfungen durchgeführt werden.

5.1.1 Gesamtarchitektur

Das Werkzeug *JastAdd* wurde mit dem Ziel konzipiert, eine Attributauswertung vorzunehmen (siehe Abschnitt 2.3 auf Seite 31). Hier wird diese Attributauswertung genutzt, um den abstrakten Syntaxbaum in einen abstrakten Syntaxgraphen zu transformieren. JastAdd führt nicht den eigentlichen Parser-Vorgang durch (die Transformation von Quelltext in den konkreten/abstrakten Syntaxbaum), sondern setzt auf einen Parser auf, der einen abstrakten Syntaxbaum erzeugt. In Abbildung 5.1 ist die Parser-Generierung unter zu Hilfenahme von JastAdd gezeigt.

Die Beschreibung der von JastAdd generierten abstrakten Syntax erfolgt mittels *ast*-Dateien. Zusätzlich zu den ast-Dateien nimmt JastAdd noch *jadd*- und *jrag*-Dateien entgegen. Erstere enthalten Aspekte (siehe [AspectJ] oder [BÖH]), die in die generierten Klassen hineingewebt werden. Innerhalb der *jrag*-Dateien können Attributauswertungen mittels einer deklarativen Sprache vorgenommen werden. Diese werden zu Aspekten transformiert und ebenfalls eingewebt. Auf diese soll nicht näher eingegangen werden, da sie nicht verwendet werden.

An den Parser-Generator muss eine Bedingung gestellt werden: er muss das Einbringen von eigenem Quelltext unterstützen, um so die generierten JastAdd-Klassen verwenden zu können. Manche Parser-Generatoren erlauben innerhalb der Spezifikation Quelltext einzubringen (wie der Parser-Generator Beaver [Beaver]), andere hingegen generieren ein Framework (wie SableCC [GAG]), in dem eigener Quelltext eingebracht werden kann. Dieser Quelltext dient dazu, den konkreten Syntaxbaum, der immer von dem Parser aufgebaut wird, in den abstrakten zu überführen. Das ist meist recht einfach, da lediglich ein paar Knoten ausgelassen werden und die Struktur geringfügig geändert

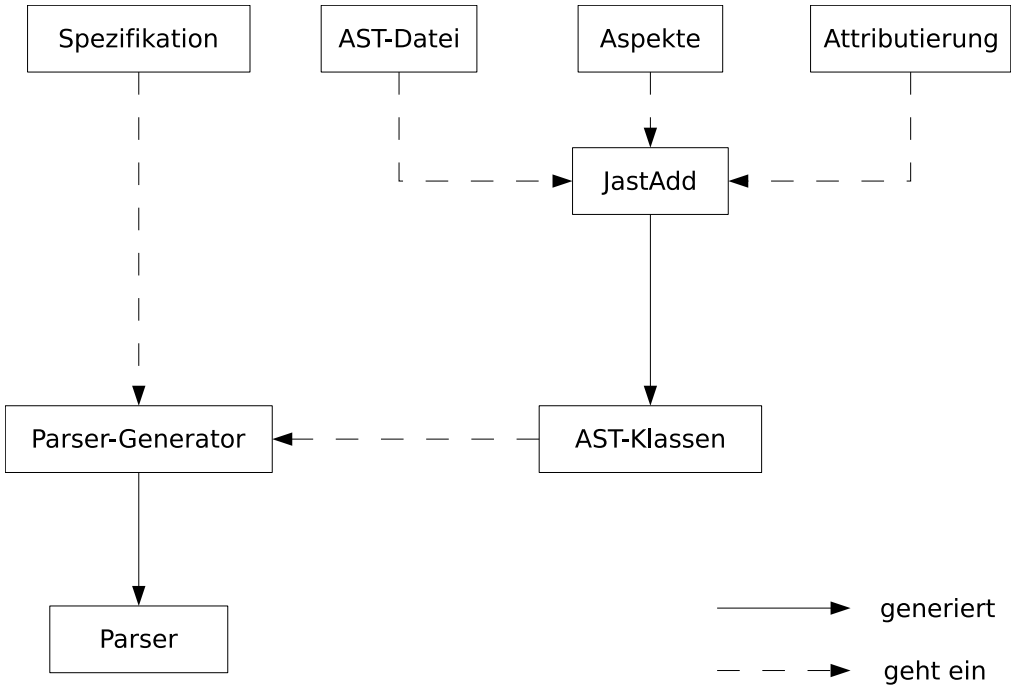


Abbildung 5.1: Allgemeine Architektur der Parserkonstruktion mit JastAdd

wird. Oft beschränkt sich diese Transformation auf das "Zusammenbauen" der Instanzen des abstrakten Syntaxbaumes. An dieser Stelle werden noch keine semantischen Prüfungen durchgeführt. Der Quelltext für die Erzeugung des abstrakten Syntaxgraphen steckt in den von JastAdd generierten Klassen (mittels Aspekte eingebunden). Das heißt, dass der abstrakte Syntaxbaum die entsprechende Transformationsmethode mitbringt, die lediglich ausgeführt werden muss. Die eingewebten Methoden prüfen die Semantik und erzeugen den Graphen.

Nachdem der Parser generiert wurde, kann diesem einen Strom von Morphemen zugeführt werden. Daraus baut der Parser einen abstrakten Syntaxbaum, der Instanz der von JastAdd generierten abstrakten Syntax ist. Die generierte abstrakte Syntax stellt ein Metamodell dar. In dieser Arbeit wird der abstrakte Syntaxbaum einer Transformation zu einem abstrakten Syntaxgraphen unterzogen. Theoretisch kann man mit JastAdd ein Metamodell definieren, das auch für diesen Zweck benutzt werden kann. Aber hier wird manuell eine abstrakte Syntax verwendet, die wiederum ihr eigenes Metamodell mit sich bringt. Die Transformation des abstrakten Syntaxbaumes in den abstrakten Syntaxgraphen erfolgt mit Methoden, die im von JastAdd generierten Metamodell eingewebt wurden. Das Zusammenspiel der einzelnen Komponenten ist in Abbildung 5.2 dargestellt.

5.1.2 Spezifikation der abstrakten Syntax

Die Spezifikation der abstrakten Syntax erfolgt mittels JastAdd innerhalb von *ast*-Dateien. Die Syntax orientiert sich dabei an EBNF-Regeln, die für Grammatikspezifikationen verwendet werden. Eine *ast*-Datei besteht aus einer Menge von Regeln, die

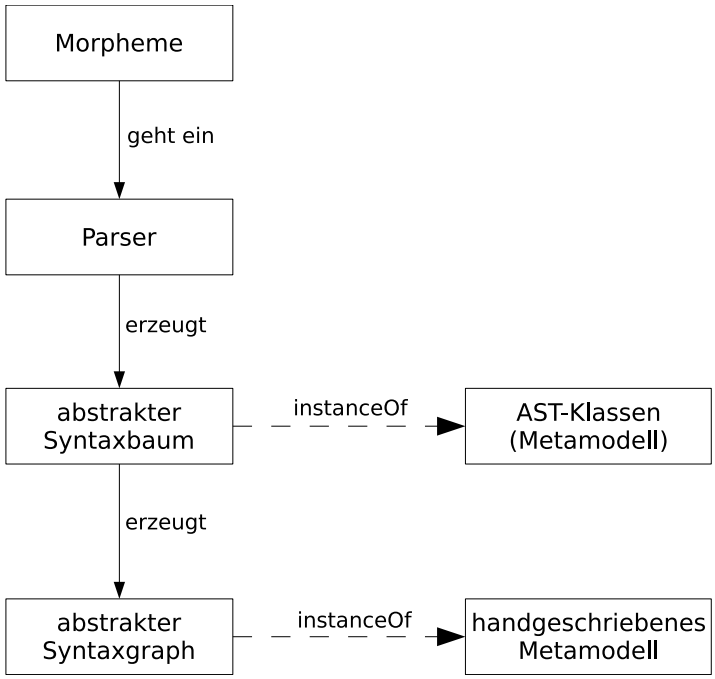


Abbildung 5.2: Transformation des abstrakten Syntaxbaumes in einen abstrakten Syntaxgraphen

eine linke und eine rechte Seite haben. Die linke Seite repräsentiert ein Nichtterminal, auf der rechten Seite können sowohl Terminale als auch Nichtterminale stehen. Die Nichtterminale repräsentieren Typen. Durch die Regeln wird eine Typhierarchie aufgebaut (siehe [GSCK]), die später in einer Klassenhierarchie endet. Die Terminale werden in <und >eingeschlossen. Sie können durch eine Typangabe näher spezifiziert werden. Diese Typen stellen die Verbindung zum Parser her, da die Typen der Terminale den Morphemklassen entsprechen, die der Parser verwendet. Wenn kein Typ angegeben ist, wird implizit der Typ `String` angenommen. Nichtterminale auf der rechten Seite können optional sein, wobei dies durch eckige Klammern [und] ausgedrückt wird. Auch der Stern * kann verwendet werden, um anzugeben, dass diese Elemente beliebig oft vorkommen dürfen.

Jedem Nichtterminal auf der rechten Seite kann ein Name zugewiesen werden, der vor der Typangabe mittels des Doppelpunktes : getrennt wird.

Nichtterminale können zudem als `abstract` definiert werden, was sich im generierten Quelltext durch abstrakte Klassen ausdrückt. Zur Veranschaulichung sollen ein paar Beispiele aus der *ast*-Datei für die Sprache PL0 gezeigt werden.

Beispiele

```
abstract ExecutionUnitAS ::=
ConstantAS* VariableAS* ProcedureAS* StatementAS
```

In dieser Regel wird das Nichtterminal `ExecutionUnitAS` definiert, was abstrakt gekennzeichnet wird. Dieses Nichtterminal setzt sich zusammen aus den Nichtterminalen

`ConstantAS`, `VariableAS`, `ProcedureAS` und `StatementAS`, wobei die ersten drei genannten beliebig oft vorkommen dürfen, während `StatementAS` genau einmal vorkommen muss.

```
BinaryOperatorAS:OperatorAS ::=
LeftSide:ExpressionAS <OperatorName:Symbol> RightSide:ExpressionAS
```

Mit dieser Regel wird das Nichtterminal `BinaryOperatorAS` definiert, es erbt vom Nichtterminal `OperatorAS` (zur Erinnerung: Nichtterminale sind Typen und bilden eine Typhierarchie). `BinaryOperatorAS` besteht aus zwei Nichtterminalen `ExpressionAS` und einem Terminal mit dem Namen `OperatorName` und dem Typen `Symbol` (`Symbol` ist eine Klasse von Beaver, die die Morpheme repräsentiert). Um die beiden Nichtterminale zu unterscheiden, bekommen sie die Namen `LeftSide` und `RightSide`.

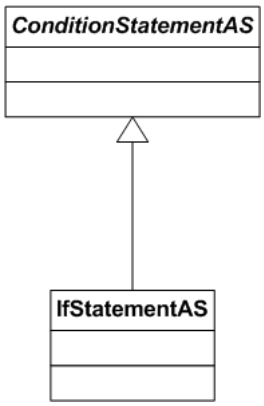
5.1.3 Generierung der AST-Klassen

Nachdem eine abstrakte Syntax definiert wurde, kann JastAdd daraus eine Menge von Java-Klassen erzeugen. Die Abbildung der Spezifikation auf die Klassen wird hier erläutert.

Jedes Nichtterminal wird zu einer Klasse. Die Nichtterminale, die mit `abstract` gekennzeichnet sind, werden zu abstrakten Klassen. Wenn einem Nichtterminal auf der linken Seite einer Regel ein Typ zugewiesen wird, dann wird die Klasse des für das Nichtterminal generierten Klasse Unterklasse des Typs. Dazu ein Beispiel. Betrachtet werde die folgende Regel:

```
IfStatementAS : ConditionStatementAS;
```

Aus dieser Regel ergibt sich folgende Klassenhierarchie:



Die Elemente auf der rechten Seite einer Regel werden sozusagen die Attribute der aus der linken Seite erzeugten Klasse. Im Prinzip werden keine Attribute generiert, statt dessen werden intern Listen von Knoten verwaltet. Aber auf die Elemente kann durch *getter*- und *setter*-Methoden zugegriffen werden. Das Namensschema setzt sich folgendermaßen zusammen: get*NameDesElements* bzw. set*NameDesElements*. Betrachtet werde folgendes Beispiel:

```
abstract ConditionStatementAS : StatementAS ::= ConditionAS StatementAS;
```

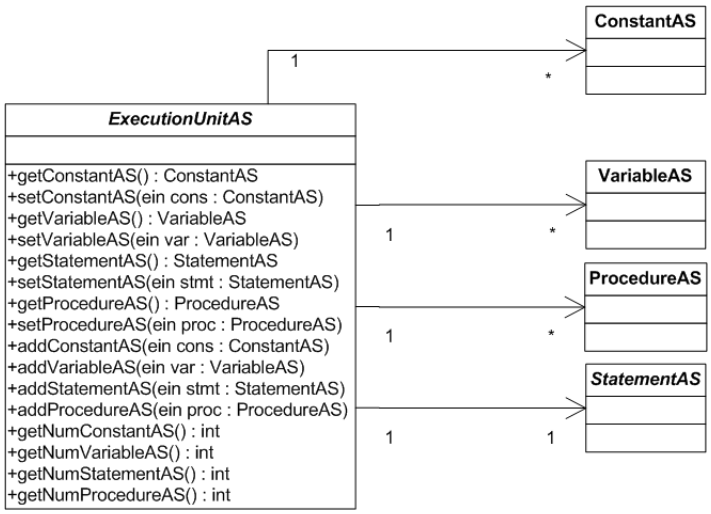


Abbildung 5.3: Generierte Klassen für die ExecutionUnit

Um auf die Elemente `ConditionAS` und `StatementAS` zugreifen zu können, können die Methoden `getConditionAS()` und `getStatementAS()` aufgerufen werden. Wenn die Elemente Namen haben, muss der Name in die `get`- bzw. `set`-Methode einfließen. Alle Elemente mit einem Stern * werden zu Listen. Die Anzahl der Elemente in einer Liste, kann durch `getNum`*NameDesElements*`():int` zurückgegeben werden, wobei der Rückgabewert ein Integer ist. Auf das einzelne Element kann mit der Methode `get`*NameDesElements*`(int index)` zugegriffen werden. Um Elemente einer Liste hinzuzufügen, gibt es noch die Methode `add`*NameDesElements*`(Elementtyp elem)`.

Für die optionalen Elemente wird eine Methode `has`*NameDesElements*`():boolean` generiert, die angibt, ob das Element verfügbar ist. Eine Besonderheit gilt es dabei zu beachten. Wenn das Element nicht gesetzt wurde und es mit der entsprechenden `get`-Methode abgefragt wird, wird eine `NullPointerException` generiert. Wenn das Element mit `null` belegt wird, dann gibt die `has`-Methode `true` zurück, aber während des Zugriffs mit der `get`-Methode wird trotzdem eine `NullPointerException` erzeugt. So sollte man nie die `set`-Methode eines optionalen Elements mit `null` aufrufen.

Zum Abschluss soll ein größeres Beispiel gegeben werden. Betrachtet werden die folgendenn Regeln:

```
abstract ExecutionUnitAS ::=
ConstrantAS* VariableAS* ProcedureAS* StatementAS*

abstract OperatorAS:ExpressionAS;

BinaryOperatorAS:OperatorAS ::=
LeftSide:ExpressionAS <OperatorName:Symbol> RightSide:ExpressionAS
```

Für die erste Regel ergibt sich die in Abbildung 5.3 dargestellte Klassenstruktur.

Für die dritte und zweite Regel ergibt sich die in Abbildung 5.4 dargestellte Klassenstruktur.

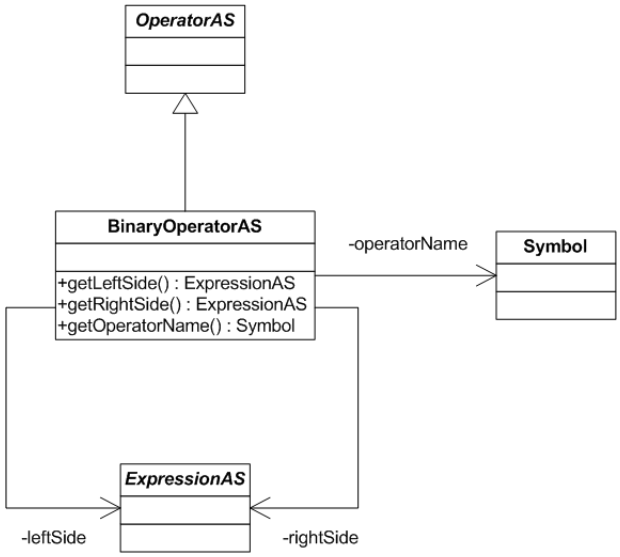


Abbildung 5.4: Generierte Klassen für die binäre Operation

Für die optionalen Elemente soll die nächste Regel als Beispiel dienen.

```
A ::= B [C] D
```

Die dazugehörige Klassenstruktur ist in Abbildung 5.5 dargestellt.

5.1.4 Der Parser-Generator Beaver

Der Parser-Generator Beaver wird in [JAPAR] als kompatibler Parser-Generator angegeben, weshalb er für die Implementierung von PL0 genutzt werden soll. Beaver ist sehr leicht zu verstehen, so dass auf die Webseite [Beaver] als Dokumentation verwiesen wird. Beaver generiert Parser für LALR-Grammatiken, wodurch er für die Implementierung für eine große Sprachklasse geeignet ist.

Innerhalb der Beaver-Spezifikationsdatei können keine Morpheme spezifiziert werden. Aus diesem Grund benötigt Beaver den Lexer-Generator *JFlex* [JFlex]. Die Zusammenarbeit zwischen beiden klappt sehr gut. Da sich die Beaver-Spezifikation mit eigenem Quelltext anreichern lässt, eignet er sich für die Verknüpfung mit JastAdd.

5.1.5 Aspekte in JastAdd

In diesem Abschnitt soll näher auf die Aspekte eingegangen werden, die in JastAdd spezifiziert werden können. Zunächst wird kurz die Syntax beschrieben, anschließend werden kleine Verwendungsbeispiele gegeben und zum Schluss noch einen Hinweis für den Umgang der Aspekte.

Syntax der Aspekte

Im Abschnitt 5.1 auf Seite 87 wurde erklärt, dass JastAdd einen Mechanismus zur Verfügung stellt, mit dem es möglich ist, Aspekte (spezifiziert in *jadd*-Dateien) in die generierten Klassen einzuweben.

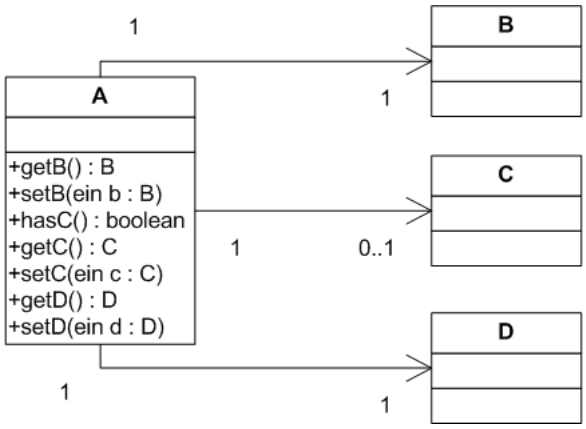


Abbildung 5.5: Generierte Klassen für optionales Element

Mit Hilfe dieser Aspektdateien können Methoden und Attribute den generierten Klassen hinzugefügt werden. Jede Aspekt-Datei besitzt folgenden grundlegenden Aufbau

```
public aspect Aspektname {
          Definition von Attributen und Methoden
}
```

Der `Aspektname` dient lediglich zur einfachen Identifizierung. Auf ihn wird nicht verwiesen. Attribute und Methoden können hinzugefügt werden, indem sie wie im normalen Java-Quelltext geschrieben werden. Es muss lediglich an einer Stelle angegeben werden, in welche Klasse das Attribut oder die Methode eingefügt werden soll. Die Syntax sieht folgendermaßen aus (zuerst wird die Definition von Attributen, dann die von Methoden gezeigt):

```
Zugriffsart Datentyp Klassenname.Attributname (optionale Initialisierung)

Zugriffsart Rückgabetyp Klassenname.Methodenname(Parameter)
```

Die `Zugriffsart` kann entweder `public`, `private` oder `protected` sein. Auch können alle andere Spezifizierer wie `abstract`, `final` oder `static` benutzt werden. Der `Datentyp` entspricht einem normalen Datentypen in Java. `Klassenname` gibt an, in welche Klasse das Attribut oder die Methode eingefügt werden soll. `Attributname` und `Methodenname` stehen für kennzeichnenden Namen. Die `Parameter` entsprechen denen von Java und die `optionale Initialisierung` ebenfalls.

Für die Programmierung mit Aspekten in JastAdd gibt es eine Einschränkung: die *Generics* aus Java1.5 können leider nicht verwendet werden.

Verwendung von Aspekten

Der Sinn der Aspekte besteht darin, einen Belang, der sich über mehrere Klassen erstreckt, in einer einzigen Datei unterzubringen. So ist es beispielsweise möglich, in einen Aspekt eine Namensauflösung durchzuführen, in einem anderen eine Typprüfung usw. Da oft alle Klassen der abstrakten Syntax betroffen sind, ist dies ein gutes Mittel, um die Implementierung dieser Belange übersichtlicher zu machen.

Hinweis für den Umgang mit den Aspektdateien

Der Quelltext der Aspektdateien wird von JastAdd automatisch in die generierten Klassen eingewebt. Während des Schreiben der Aspekte können sich Syntaxfehler einschleichen (der Eclipse-Editor ist hier leider nur beschränkt benutzbar), so dass diese auf die generierten Klassen übertragen werden. In den generierten Klassen wird man in Eclipse durch rote Markierungen darauf aufmerksam gemacht, dass Fehler vorhanden sind. Man sollte diese immer in den Aspekt-Dateien korrigieren, da alle Änderungen an den generierten Klassen sonst verlorengehen würden.

5.2 PL0-Implementierung

Um den OCL-Parser zu implementieren, gibt es verschiedene Möglichkeiten. Das Werkzeug *JastAdd* trat dabei in die engere Wahl. Um sich mit JastAdd bekanntzumachen und die Arbeitsweise von JastAdd zu verstehen (auch die Fallstricke herauszubekommen), wurde probeweise die kleine Sprache PL0 implementiert (siehe Kapitel 3, Seite 43). PL0 ist so klein, dass der Aufwand begrenzt werden konnte. Die PL0-Implementierung bildet darüber hinaus einen relativ überschaubaren Einblick in die Arbeitsweise der gesamten Architektur, die für den OCL-Parser ähnlich ist.

Zunächst wird die Architektur des gesamten Parsers vorgestellt, insbesondere die Verflechtung mit JastAdd. Anschließend werden die zwei Phasen betrachtet, in denen der abstrakte Syntaxgraph aufgebaut wird. Und zum Schluss werden noch ein paar Besonderheiten der Implementierung erläutert.

5.2.1 Architektur und Ziel des PL0-Parsers

Der PL0-Parser soll in der Lage sein, ein PL0-Programm einzulesen, die statische Semantik zu prüfen (siehe Abschnitt 3.5.1, Seite 56) und eine Transformation in einen abstrakten Syntaxgraphen vorzunehmen, der eine Instanz der abstrakten Syntax von PL0 ist (siehe Abschnitt 3.4, Seite 50).

Ein Schema für die Erzeugung des PL0-Parsers ist in Abbildung 5.6 dargestellt. JastAdd bekommt eine Beschreibung der abstrakten Syntax als *ast*-Datei übergeben. Zudem erhält JastAdd zwei Aspekte, einen für die Namensauflösung und einen für das Bauen des abstrakten Syntaxgraphen. JastAdd generiert daraus eine Klassenhierarchie, die die abstrakte Syntax bildet. Auf diese wird in der Grammatikdatei für Beaver Bezug genommen. Da innerhalb von Beaver keine Morpheme definiert werden können, muss der Lexer-Generator JFlex benutzt werden. Er generiert aus der PL0-Morphemspezifikation einen Lexer. Die Morphemnamen fließen in die Beaver-Grammatikspezifikation ein. Aus dieser Grammatikdatei wird dann der PL0-Parser generiert.

Der PL0-Parser nimmt einen Strom von Morphemen entgegen (im Grunde ein PL0-Programm) und baut daraus einen abstrakten Syntaxbaum, der Instanz des von JastAdd generierten Metamodells ist. Danach wird die statische Semantik geprüft und hinzugefügte Referenzen aufgelöst (siehe unten *Namensauflösung*). Nachdem die Prüfung der statischen Semantik erfolgreich war, wird ein abstrakter Syntaxgraph einer manuellen erstellten abstrakten Syntax gebaut. Zusammengefasst ist diese Vorgehensweise in Abbildung 5.7.

Die PL0-Implementierung wurde mit der Entwicklungsplattform *Eclipse* [Eclipse] erstellt und für diese ein eigenes Eclipse-Projekt erzeugt. Dieses Projekt besteht aus Spezifikationsdateien für die verwendeten Werkzeuge, der abstrakten Syntax, ein paar wenigen Hilfsklassen und den generierten JastAdd-Klassen.

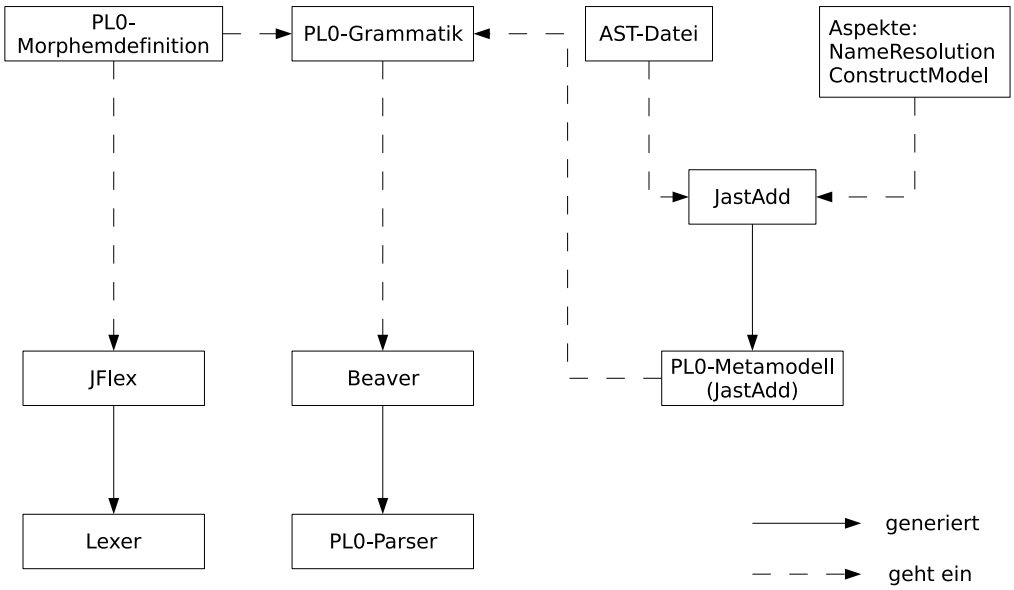


Abbildung 5.6: Schema für die Erzeugung des PL0-Parsers

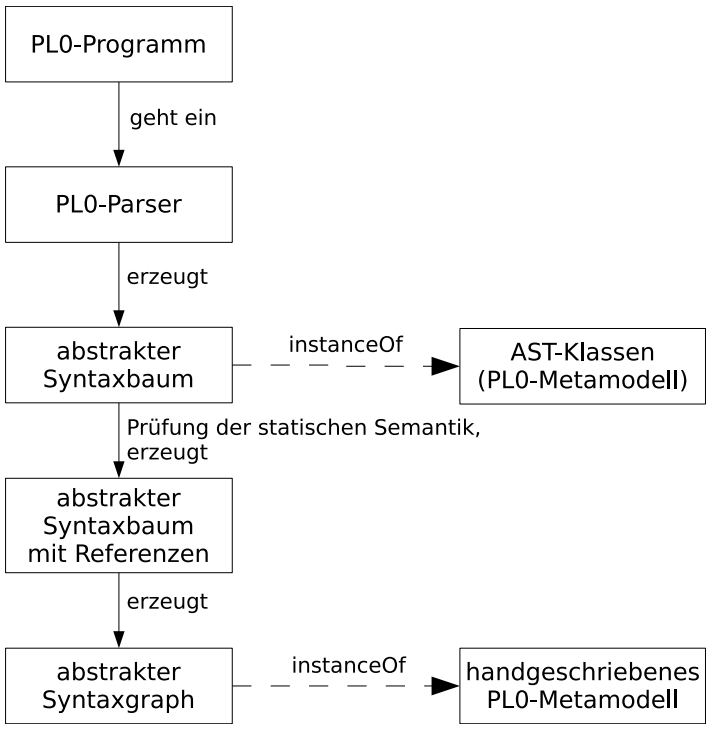


Abbildung 5.7: Prinzipielle Vorgehensweise zur Transformation in den abstrakten Syntaxgraphen

Datei	Bedeutung
pl0.flex	Definiert die Morpheme, Eingabe für JFlex
pl0.parser	Definiert konkrete Syntax, Quelltext für die Erzeugung des abstrakten Syntaxbaumes, Eingabe für Beaver
pl0.ast	Definiert die abstrakte Syntax, Eingabe für JastAdd

Tabelle 5.1: Spezifikationsdateien und ihre Bedeutung

Im Ordner `spec` befinden sich die Beaver- und JFlex-Spezifikationen. In Tabelle 5.1 sind alle Spezifikationsdateien mit ihrer Bedeutung aufgeführt.

Der aufmerksame Leser wird sich an dieser Stelle fragen, warum es offensichtlich zwei abstrakte Syntaxe gibt: eine manuell geschriebene und eine, die von JastAdd generiert wird (die Klassen dieser abstrakten Syntax besitzen das Postfix `AS`). Dieses Vorgehen besitzt mehrere Gründe. Die von Hand implementierte abstrakte Syntax besteht aus einfachen Klassen, die lediglich Attribute und ihre `getter`- und `setter`-Methoden besitzen. Die von JastAdd generierten Klassen bringen noch zusätzlichen Quelltext mit, der für die weitere Verarbeitung des am Ende erstellten abstrakten Syntaxgraphen unnütz wäre. Es kann durchaus sein, dass die abstrakte Syntax, in die ein Quelltext überführt werden soll, schon existiert wie im Fall OCL (siehe Abschnitt 6 auf Seite 99). Die abstrakte Syntax heißt dort *Essential OCL* und wurde im Rahmen der Belegarbeit [Braeuer] bereits implementiert. Das heißt, es muss von einer Instanz der JastAdd generierten Klassenhierarchie (oder auch Metamodells) eine Transformation in eine andere Instanz (eines anderen Metamodells) vorgenommen werden.

5.2.2 Phasen der Transformation

Die von JastAdd generierten Klassen können in der PL0-Implementierung lediglich einen abstrakten Syntaxbaum bilden, da die Blätter nur Morpheme enthalten und keine Referenzen auf andere Strukturen innerhalb des Baumes verweisen. Die Transformation des abstrakten Syntaxbaumes in einen abstrakten Syntaxgraphen (Instanz der handgeschriebenen abstrakten Syntax) erfolgt mittels der Aspekt-Dateien (im Ordner `spec/jastAddFiles`). Die Transformation verläuft dabei in zwei Phasen. Zuerst wird eine *Namensauflösung* am abstrakten Syntaxbaum vorgenommen und anschließend die Transformation in den abstrakten Syntaxgraphen.

Namensauflösung

Die Namensauflösung befindet sich in der Datei `NameResolution.jadd`. Allen von JastAdd generierten Klassen, die ein Morphem beinhalten, bekommen ein zusätzliches Attribut aus der handgeschriebenen abstrakten Syntax zugewiesen. Beispielsweise besitzt die Klasse `InputStatementAS` ein Morphem. Dieser Klasse wird das Attribut `Variable variable` hinzugefügt. Die Klasse `ExecutionUnitAS` bekommt zusätzlich drei Listen von Konstanten, Variablen und Prozeduren (genaueres zu den Prozeduren siehe Abschnitt 5.2.3, Seite 97). Diese hinzugefügten Attribute bekommen während der Namensauflösung Referenzen zugeordnet.

Jede Klasse der abstrakten Syntax von JastAdd erhält eine Methode

```
public void makeNameResolution(Environment env)
```

, die die Namensauflösung (oder Referenzauflösung) durchführt. Die Umgebung (Parameter `env`) speichert zuvor berechnete Elemente wie Prozeduren, Variablen und Konstanten (siehe Abschnitt 3.5.1 auf Seite 56). Die Berechnungsmethoden rufen rekursiv ihre Kindelemente auf. Dazu ein kleines Beispiel. Die Definition des Nichtterminals `IfConditionAS` sieht in der *ast*-Datei so aus:

```
abstract ConditionStatementAS : StatementAS ::= ConditionAS StatementAS;
IfStatementAS : ConditionStatementAS;
```

Innerhalb der Berechnungsmethode wird die `makeNameResolution`-Methode von `ConditionAS` aufgerufen und anschließend von `StatementAS`. So wird durch den gesamten Baum bis zu den Blättern traversiert. An den Blättern werden die neu hinzugefügten Attribute gesetzt. Während der Namensauflösung wird die statische Semantik (siehe 3.5.2 auf Seite 58) geprüft.

Aufbau des abstrakten Syntaxgraphen

Nachdem die Namensauflösung erfolgreich verlief, muss der abstrakte Syntaxgraph aufgebaut werden. In der Datei `ConstructModelAspect.jadd` befindet sich diese Transformation. Jede von JastAdd generierte Klasse erhält eine `constructModel()`-Methode. Diese Methode ruft rekursiv die Kindelemente des abstrakten Syntaxbaumes auf und berechnet den abstrakten Syntaxgraphen (als Instanz der handgeschriebenen abstrakten Syntax). Die Methode gibt immer ein Element der handgeschriebenen abstrakten Syntax zurück. Das oberste Element des Graphen bildet die Klasse `Program`. Die Blattknoten des abstrakten Syntaxbaumes geben nur die Referenzen zurück, die im Laufe der Namensauflösung berechnet wurden.

5.2.3 Besonderheiten der Implementierung

Im Folgenden soll auf ein paar Besonderheiten der Implementierung eingegangen werden.

Die Klasse BlockContainerAS

Wie schon in 3.4.1 (Seite 51) beschrieben, stellt die Klasse `BlockContainerAS` lediglich eine Hilfsklasse dar. Sie wird innerhalb der *ast*-Datei definiert und ausschließlich in der Beaver-Spezifikationsdatei verwendet. Diese Klasse ist notwendig, weil die Grammatik das Nichtterminal `BLOCK` einführt. Der abstrakten Syntaxbaum, der von dem Beaver-generierten Parser aufgebaut wird, enthält keine Instanz der Klasse `BlockContainerAS`.

Der DotGraph-Aspekt

Im Verzeichnis `spec/jastAddFiles` liegt der Aspekt `DotGraphAspect.jadd`. Dieser fügt den von JastAdd generierten Klasse eine Methode `buildDotEdges` hinzu, die eine *dot*-Grafik (siehe [Dot]) aus dem abstrakten Syntaxbaum aufbaut. Dieser Aspekt ist aus der Motivation heraus entstanden, prüfen zu können, ob der Parser korrekt arbeitet.

6 OCL-Parser

In diesem Kapitel wird das Hauptergebnis dieses Beleg vorgestellt. Zunächst werden die Entwurfsentscheidungen dargelegt, um einen Eindruck davon zu vermitteln, warum der Parser in der jetzigen Form vorliegt. Anschließend wird das Vorgehen für die Implementierung des Parser aufgezeigt. Den Schluss des Kapitels bilden die semantischen Prüfungen, die umgesetzt wurden.

Das Pivotmodell führt im Gegensatz zum UML-Metamodell teilweise andere Begriffe ein. Das Konstrukt *Klasse* im UML-Metamodell wird zum Konstrukt *Type* im Pivotmodell, das Konstrukt *Attribut* wird zum Konstrukt *Proptery*. Im Text steht meist *Type* oder *Property* in Klammern, da sich alle Beispiel auf UML-Modelle beziehen.

Die abstrakte OCL-Syntax wurde im Rahmen von [Braeuer] ein wenig modifiziert, um sie dem Pivotmodell anzupassen.

6.1 Entwurfsentscheidungen

Der OCL-Parser wurde mit den zwei Werkzeugen *Erweitertes SableCC* (siehe dazu auch Kapitel 4 auf Seite 61) und *JastAdd* (siehe dazu Kapitel 5 auf Seite 87) implementiert. Warum diese Werkzeuge gewählt wurden, wird im ersten Abschnitt erläutert. Im darauffolgenden Abschnitt wird die Architektur des Parsers vorgestellt.

6.1.1 Wahl der Software-Werkzeuge

Wie im Kapitel 3 (Seite 43) beschrieben, wurde die Sprache PL0 mehrere Male implementiert, um einen Überblick über passende Software-Werkzeuge zu bekommen. In die engere Wahl traten dabei die SableCC-Erweiterung (siehe Kapitel 4 auf Seite 61) und der Parser-Generator Beaver (siehe [Beaver]) oder AntLR (siehe [Antlr]) in Verbindung mit dem Attributauswerter JastAdd (siehe [JASTWeb]).

Anfangs stand lediglich zur Diskussion, ob die SableCC-Erweiterung Verwendung finden soll oder JastAdd mit Beaver oder AntLR.

Mit Hilfe der SableCC-Erweiterung ist es möglich, die semantischen Prüfungen (im Englischen *wellformedness rules*), die vom OCL-Standard [OCL2] (Seite 33 bis 60) gefordert werden, zu implementieren. Für diesen Ansatz sprach, dass der bestehende Parser mit diesem Werkzeug geschrieben wurde und so war das Risiko eines Fehlschlages geringer. Gegen die SableCC-Erweiterung sprach, dass manche Erweiterungen (wie beispielsweise `#customheritage` und `#maketree`) nicht ganz einfach zu verstehen sind. Außerdem wäre der OCL-Parser abhängig vom einem Werkzeug, das nicht weitergepflegt wird (der Standard SableCC-Parser-Generator wird dagegen weiterentwickelt).

Das Werkzeug JastAdd dagegen wird von der Lund Universität gepflegt und es verfolgt einen neuen Ansatz. Es nutzt einen Parser-Generator, der lediglich in der Lage sein muss, Bezug zu den von JastAdd generierten Klassen herzustellen, und der Nutzer muss eigenen Quelltext einfügen können, um die JastAdd-Klassen zu instanziieren. JastAdd führt zwei Arten der Attributauswertung ein (siehe [JAST], Seite 8 bis 18): deklarativ und imperativ. Die imperative Möglichkeit wird mittels Aspekte umgesetzt (siehe

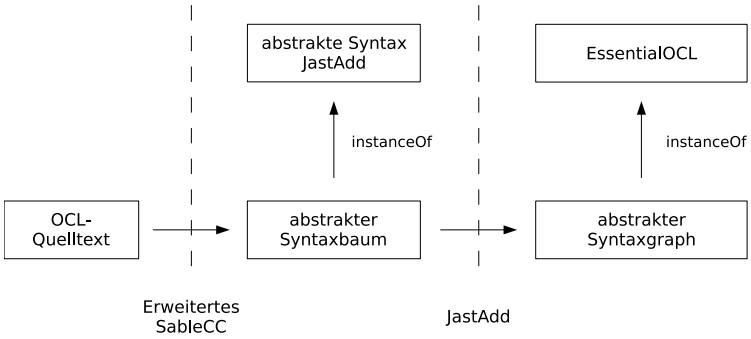


Abbildung 6.1: Phasen des gesamten OCL-Parser-Vorgangs

Kapitel 5 auf Seite 87). Sie erlaubt es Quelltext, der semantisch zusammengehört, zu zentralisieren. Die Aspekte werden in die von JastAdd generierten Klassen eingewebt. Da sich die semantischen Prüfungen über diese Klassen in der Regel verstreuen, kann eine leichtere Einarbeitung und Wartung erreicht werden.

Als kompatibler Parsergenerator wird in [JAPAR] unter anderem Beaver erwähnt. In Abschnitt 5.1.4 auf Seite 92 wurde Beaver im Rahmen der PL0-Implementierung vorgestellt. PL0 ließ sich problemlos mit Beaver umsetzen.

Im weiteren Verlauf der Arbeit fiel die Entscheidung zugunsten von JastAdd und Beaver aus (aus besagten Gründen). Während der Portierung der OCL-Grammatik des ursprünglichen OCL-Parsers stellte sich jedoch heraus, dass Beaver einen Fehler aufweist, so dass Konflikte innerhalb der Grammatik aufgezeigt wurden, die nicht existierten. In diesem Moment kam der Parser-Generator AntLR ins Spiel. AntLR besitzt eine große Anhängerschaft und man kann davon ausgehen, dass es weiterentwickelt und -gewartet wird. Es stellte sich heraus, dass AntLR lediglich LL-Grammatiken verarbeiten kann, was zur Folge gehabt hätte, die OCL-Grammatik grundlegend zu verändern.

Aus diesem Grund wurden die ursprünglichen getrennten Gedanken zusammengefügt. Als Parser-Generator kommt die SableCC-Erweiterung zum Einsatz und die semantische Prüfung wird mittels JastAdd umgesetzt. Die Abhängigkeit von der SableCC-Erweiterung hält sich in Grenzen, da der Parser lediglich zum Aufbau des abstrakten Syntaxbaumes dient. Die Implementierung dieses Aufbaus ist mit vielen Parser-Generatoren leicht zu bewältigen. Es gibt allerdings zwei Aufgaben, die der Parser in diesem Fall neben dem Aufbau des abstrakten Syntaxbaumes durchführt (siehe Abschnitt 6.2.6).

6.1.2 Aufbau des Parsers

Die Aufgabe des OCL-Parsers besteht darin, einen Text - geschrieben in der Sprache OCL - in eine Instanz von *EssentialOCL* zu transformieren. EssentialOCL stellt eine Teilmenge der in [OCL2] spezifizierten abstrakten Syntax dar. Auch wurde diese abstrakte Syntax an das Pivotmodell angepasst (siehe [Braeuer]). In EssentialOCL fehlen beispielsweise die Konstrukte `MessageExp` und `StateExp`.

Die Transformation eines OCL-Ausdruckes in eine Instanz von EssentialOCL verläuft in zwei Phasen, wobei die erste Phase ein Zwischenergebnis hervorbringt (siehe Abbildung 6.1).

Zunächst wird der OCL-Quelltext mit dem Parser (generiert von der SableCC-Erweiterung) in einen abstrakten Syntax**baum** transformiert. Dieser Baum ist Instanz der Klassenhierarchie, die von JastAdd als abstrakte Syntax generiert wird. Der abstrakte Syntaxbaum wird anschließend in einen abstrakten Syntax**graphen** transformiert, der Instanz von EssentialOCL ist. Diese Transformation wird von Methoden durchgeführt, die sich in den von JastAdd generierten Klassen wiederfinden. Diese Methoden führen die semantischen Prüfungen durch (melden gegebenenfalls Fehler) und bauen den abstrakten Syntaxgraphen auf. Diese Methoden werden in Aspektdateien definiert und von JastAdd eingewebt.

6.2 Vorgehen

Die Implementierung des OCL-Parsers gliedert sich in einzelne Teilschritte:

1. Spezifikation der Morpheme und der Grammatik für die Sprache OCL
2. Aufstellen der abstrakten Syntax in JastAdd
3. Grammatikspezifikation mit Typen anreichern
4. Anpassung der SableCC-Erweiterung
5. Generierung des OCL-Parsers
6. Implementieren der Transformation des konkreten Syntaxbaumes in einen abstrakten
7. Implementierung der semantischen Prüfung und Transformation des abstrakten Syntaxbaumes in einen abstrakten Syntaxgraphen

Die Punkte 1 bis 6 werden in diesem Abschnitt vorgestellt. Punkt 7 ist dagegen ein eigener Abschnitt gewidmet. In einem letzten Abschnitt wird kurz darauf eingegangen, wie der OCL-Parser in das *Dresdner OCL-Toolkit* eingebunden wird.

6.2.1 Grammatikspezifikation

Die OCL-Spezifikaton definiert auf den Seiten 61 bis 94 in [OCL2] eine konkrete Syntax. Diese lässt sich nicht als Eingabe für einen Parser-Generator verwenden, da sie Mehrdeutigkeiten enthält und es können mit der Grammatik Ausdrücke erzeugt werden, die nicht gültig sind. Beispielsweise wird das Nichtterminal `OperationCallExpCS` in einer Alternativen folgendermaßen definiert:

```
OperationCallExpCS := OCLExpressionCS[1] simpleName OCLExpressionCS[2]
```

Damit wäre es theoretisch müglich den Ausdruck

```
a b a
```

zu erzeugen, der offensichtlich keinen gültigen OCL-Ausdruck darstellt. (Dieser Ausdruck lässt sich erzeugen, indem beide Nichtterminale `OCLExpressionCS` zu dem Nichtterminal `AttributeCallExpCS` abgeleitet werden, das Symbol `b` entsteht durch das Morphem `simpleName`.) Neben den Grammatikregeln gibt es Angaben zur Attributauswertung und Regeln für das Auflösen von Mehrdeutigkeiten (in der Spezifikation mit *Disambiguating rules* bezeichnet). Die Attributauswertung spielt insbesondere während

der semantischen Analyse eine bedeutende Rolle. Die Regeln für die Auflösung von Mehrdeutigkeiten können mit einer Transformation der Grammatik in großen Teilen implizit eingebracht werden. (Nebenbei bemerkt: mit der oben gezeigten Regel werden die mathematischen Ausdrücke beschrieben, wobei `simpleName` zu einem Operationszeichen abgeleitet wird.) In [ANS] wurde eine Transformation der angegebenen OCL-Grammatik bereits vorgenommen. Diese Grammatik dient für diese Arbeit als Ausgangsbasis. Die Morphem- und Regeldefinitionen wurden noch einmal angepasst. Das Ziel dieser Anpassung bestand darin, möglichst viele Prüfungen vom Parser durchführen zu lassen, um den Aufwand der semantischen Prüfung zu reduzieren. Da die Grammatikspezifikation des ursprünglichen OCL-Parser in der Spezifikationssprache der SableCC-Erweiterung verfasst wurde, konnte diese als Ausgangsbasis dienen. Die endgültige Datei findet sich unter dem Pfad *./src/spec/ocl2.parser*.

Neben der Anpassung der Morphem- und Regeldefinitionen wurden die Typinformationen an die abstrakte Syntax angepasst (von JastAdd generiert).

Auf ein paar wenige Besonderheiten der Grammatikspezifikation wird im Folgenden näher eingegangen. Zunächst werden die Morpheme und anschließend die Grammatikregeln betrachtet.

Morpheme

Die meisten Schlüsselwörter haben die Markierung `!` erhalten, um anzuzeigen, dass sie während der Traversierung des konkreten Syntaxbaumes ausgelassen werden. Für einige Morpheme trifft dies nicht zu, obwohl sie theoretisch im konkreten Syntaxbaum keine wichtige Rolle einnehmen. Dazu gehören beispielsweise die Morpheme `pre`, `body` und `post`. Diese Schlüsselwörter werden in den folgenden beiden Grammatikregeln verwendet (ohne Typinformation):

```
operation_constraint_cs =
{full} op_constraint_stereotype_cs simple_name? colon ocl_expression_cs
| {empty} op_constraint_stereotype_cs colon;

op_constraint_stereotype_cs =
{pre} pre | {post} post | {body} body;
```

Die Information, um welche Bedingungsart (`pre`, `body` oder `post`) es sich handelt, könnte auch implizit durch einzelne Regeln gespeichert werden (für jede Bedingungsart eine). Das hätte zur Folge, dass es anstelle der zwei Regeln sechs Regeln geben würde. Das Nichtterminal `op_constraint_stereotype_cs` wurde eingeführt, um diesen Aufwand zu sparen.

Eine Besonderheit betrifft die Typen der Morpheme, die nicht mit dem Ausrufezeichen `!` gekennzeichnet sind. Da diese während der Traversierung des konkreten Syntaxbaumes durchlaufen werden, und dementsprechend ausgewertet werden, muss deren Auswertung einen Rückgabewert liefern. In diesem Fall werden die Informationen in ein `TokenAS`-Objekt verpackt. Die Klasse `TokenAS` wird in der abstrakten Syntax (JastAdd) definiert. Sie enthält lediglich Zeilen-, Spalten- und Wertinformationen des Morphems. Die `TokenAS`-Instanzen bilden im abstrakten Syntaxbaum die Blätter. Da die semantische Prüfung anhand dieses Baumes durchgeführt wird, müssen die Positionsdaten für die Fehlerausgabe zur Verfügung stehen, um den Nutzer bei einem Fehler auf die richtige Stelle im OCL-Ausdruck aufmerksam machen zu können.

Grammatikregeln

Die Grammatikregeln innerhalb der SableCC-Spezifikation unterscheiden sich teilweise erheblich von der konkreten Syntax der OCL-Spezifikation [OCL2]. Die Veränderungen der konkreten Syntax hin zur benutzten Grammatik sind im Anhang B auf Seite 143 ausführlich dargestellt.

Die Grammatik lässt nur OCL-Ausdrücke zu, die mit dem Schlüsselwort `package` beginnen. Mit diesem Schlüsselwort wird der Wurzelnamensraum (im Englischen *root namespace*) festgelegt, in dem sich die Modellelemente befinden, auf die sich der Ausdruck bezieht. Hier könnte Änderungsbedarf bestehen, denn in einem graphischen Werkzeug steht der Wurzelnamensraum bereits implizit fest, so dass der Benutzer einen möglichst kurzen Ausdruck schreiben möchte. Sogar die Kontextdefinition (eingeleitet durch das Schlüsselwort `context`) könnte unterlassen werden, wenn man sich vorstellt, dass die OCL-Ausdrücke in kommentarähnlichen Kästchen direkt mit dem Attribut, mit der Operation oder mit einer Klasse verbunden sind.

Innerhalb der Definition des Nichtterminals `operation_constraint_cs` gibt es die Alternative:

```
operation_constraint_cs = ...
| {empty} op_constraint_stereotype_cs colon;
```

Damit kann beispielsweise folgender Ausdruck erzeugt werden:

```
context A::method()
pre:
post: result = 5
```

Der Sinn solcher leeren Teilausdrücke besteht darin, dem Leser explizit anzuzeigen, dass der OCL-Ausdruck keine Vorbedingung hat. Nach dem Teilausdruck `pre :` kann auch ein Kommentar folgen. Solche Ausdrücke dienen nur der besseren Lesbarkeit und Vereinfachung. Während der Transformation des konkreten Syntaxbaum in den abstrakten Syntaxbaum müssen diese Ausdrücke natürlich entfernt werden.

Eine weitere Besonderheit betrifft das Nichtterminal `identifier_cs`, das folgendermaßen definiert ist:

```
identifier_cs = {simple} simple_name
      | {ocl_op_name} ocl_op_name
      | collection_type_identifier_cs;
```

Es fasst drei Arten von Morphemen zusammen. Ein Operations- oder Attributname kann einem von diesen drei Morphemarten entsprechen. Diese wurden mit dem Nichtterminal `identifier_cs` zusammengefasst. So muss nicht für jede einzelne Morphemart separate Regeln für die Attribut- und Operationsaufrufe geschaffen werden.

Die arithmetischen Ausdrücke (beginnend beim Nichtterminal `logical_exp_cs` in der Grammatik) sind nicht Bestandteil der konkreten Syntax aus [OCL2]. Stattdessen wird im Standard die merkwürdige Regel:

```
OperationCallExpCS := OCLExpressionCS[1] simpleName OCLExpressionCS[2]
```

definiert. Mit dieser Form ist es nicht möglich, Prioritäten einzuführen. Die Prioritäten werden durch die "treppenartige" Struktur der Grammatikregeln eingeführt (im Abschnitt 2.2.2 auf Seite 25 ist eine vereinfachte Form dieser "Treppenstruktur" dargestellt).

Die Definition des Nichtterminals `postfix_exp_cs` mutet zunächst sehr komplex an. Beim näheren Hinsehen stellt man sogar fest, dass sich Teile der Regeln überschneiden, so dass es sinnvoll sein könnte, diese gemeinsamen Teile in separate Regeln auszulagern. Der Grund, warum dies nicht gemacht wurde, ist die Vermeidung von zusätzlichen *Container*-Klassen in der abstrakten Syntax (JastAdd), siehe dazu auch die Klasse `BlockContainerAS` im Abschnitt 3.4.1 auf Seite 51.

6.2.2 Aufstellen der abstrakten Syntax

Nachdem die Morpheme und Grammatikregeln spezifiziert wurden, geht es darum eine Grammatik für die abstrakte Syntax mit JastAdd zu schreiben. Die Datei, in der die abstrakte Syntax spezifiziert ist, findet sich unter *./src/spec/jastadd/ocl2.ast* wieder. Alle Konstrukte/Nichtterminale, die darin definiert werden, tragen den Postfix *AS*, um sie von den Klassen der EssentialOCL zu unterscheiden. Ziel während der Formulierung der abstrakten Syntax war es, sie an EssentialOCL anzunähern, um den abstrakten Syntaxbaum relativ leicht in den abstrakten Syntaxgraphen überführen zu können. Zudem bildete EssentialOCL die Ausgangsbasis für die zu konstruierende abstrakte Syntax.

Die abstrakte Syntax kann in zwei Teile gegliedert werden. Zum einen gibt es die Konstrukte, die die Paket- und Kontextdeklarationen bilden (siehe Nichtterminale `OclFileAS`, `PackagedConstraintAS`, `ContextAS`, `ConstraintAS`, ...). Zum anderen findet sich die Klassenhierarchie um das Konstrukt `OclExpression` aus EssentialOCL in der abstrakten Syntax mit dem Nichtterminal `OclExpressionAS` wieder.

Es wurde versucht für jede Regel der Grammatik eine Entsprechung in der abstrakten Syntax zu finden und auch Subklassenbeziehungen zu nutzen. So zum Beispiel besitzt jeder Kontext (eingeleitet durch das Schlüsselwort `context`, vgl. Nichtterminal `context_declaration_cs` in der Grammatik) einen Pfadnamen (nämlich den, auf den sich der OCL-Ausdruck bezieht) unabhängig davon, ob es sich um einen OCL-Ausdruck handelt, der sich auf ein Attribut (bzw. *Property* im Pivotmodell), einer Operation oder eine Klasse (bzw. ein *Type* im Pivotmodell) bezieht. Dies spiegelt sich in der Definition des Nichtterminals `ContextAS` wieder:

```
abstract ContextAS ::= Name:PathNameAS;
```

Die Subklassen erhalten die Attribute, die in den anderen Regeln der Grammatik hinzukommen (ausführlicher ist dies in Abschnitt 6.2.3 dargestellt).

Die Parameterlisten müssen in der konkreten Syntax mit mehreren Regeln dargestellt werden. In der abstrakten Syntax werden sie oft zu einer Liste von Variablen oder Instanzen der Klasse `TokenAS`.

Wie oben bereits bemerkt, stellt das Nichtterminal `TokenAS` die Blattknoten im abstrakten Syntaxbaum dar.

Es gibt ein paar Regeln, die die Bedingung (siehe Nichtterminal `ConstraintAS` und deren Ableitungen bzw. Unterklassen) ausdrücken. Durch den Namen der Unterklassen wird die Information implizit gespeichert, um welche Bedingung es sich handelt. Beispielsweise werden alle Bedingungen, die eine Klasse (*Type*) betreffen mit Instanzen der Klasse `ClassifierConstraintAS` gefasst.

Eine besondere Regel stellt die Definition des Nichtterminals `OperationCallExpAS` dar:

```
OperationCallExpAS : FeatureCallExpAS ::=
Name:PathNameAS Argument:OclExpressionAS*
<ArrowRightExpression:boolean> <DotExpression:boolean>;
```

Die beiden booleschen Werte `ArrowRightExpression` und `DotExpression` sind hier von besonderer Bedeutung. Mit diesen beiden Attributen (zur Erinnerung: die Nichtterminale werden zu Klassen transformiert) wird gespeichert, ob es sich um einen Operationsaufruf handelt, der nach einem Punkt- oder einem Pfeiloperator aufgerufen wird. In der konkreten Syntax gibt es drei Möglichkeiten eine Operation darzustellen:

```
postfix_exp_cs = ...
| {dot_operation_call} postfix_exp_cs dot identifier_cs atpre?
  open_paren actual_parameter_list_cs? close_paren
...

| {arrowright_operation_call} postfix_exp_cs arrowright
  collection_operation
  open_paren actual_parameter_list_cs? close_paren
...


property_call_exp_cs = ...
| {parameter} path_name_cs atpre?
  open_paren actual_parameter_list_cs? close_paren
...
```

Innerhalb der semantischen Prüfung wird die Information benötigt.

6.2.3 Einführen der Typen in die Grammatikspezifikation

Nachdem die abstrakte Syntax spezifiziert wurde, muss nun die Grammatikspezifikation mit den Typen/Klassen angereichert werden, die von JastAdd aus der abstrakten Syntax generiert werden. Die SableCC-Erweiterung stellt dazu eine einfache Möglichkeit bereit (siehe Abschnitt 4.2.2 auf Seite 70).

Da die abstrakte Syntax aus der konkreten gewonnen wurde, bringt die Typannotation keine großen Probleme mit sich, denn die Typen werden den Regel zugewiesen, aus denen sie hervorgegangen sind. Dazu ein Beispiel. Betrachtet wird die folgende Regel aus der konkreten Syntax:

```
collection_literal_exp_cs<CollectionLiteralExpAS> =
collection_type_identifier_cs
         braceopen collection_literal_parts_cs? braceclose;
```

Diese Regel bekommt den Typ `CollectionLiteralExpAS` annotiert. Die Regel enthält einen Identifizierer (`identifier_cs`) und eine Liste von Kollektionsliteralen (`collection_literal_parts_cs`). Die anderen Symbole werden ausgelassen. Die Definition des Typs `CollectonLiteralExpAS` sieht in der abstrakten Syntax folgendermaßen aus:

```
CollectionLiteralExpAS: LiteralExpAS ::= Name:TokenAS
                                         CollectionLiteralPartAS*;
```

Wie zu sehen, finden sich die beiden Elemente aus der Regel der konkreten Syntax dort wieder.

Etwas schwieriger verhält es sich mit den Regeln, die keine Entsprechung in der abstrakten Syntax besitzen, wie beispielsweise für die Nichtterminale:

- `formal_parameter_list_cs`
- `actual_parameter_list_cs`

In diesen beiden Fällen wird eine Liste aus der abstrakten Syntax zurückgegeben (siehe Klasse `ocl2as.List`).

Oft kann das Schlüsselwort `#chain` (siehe Abschnitt 4.2.4 auf Seite 75) verwendet werden, da es viele Regeln gibt, die nur die Erzeugung eines Satzes "weiterleiten", wie beispielsweise folgende Regel:

```
formal_parameter_enum_cs<VariableExpAS> = comma formal_parameter_cs#chain;
```

6.2.4 Anpassung der SableCC-Erweiterung

Anpassung der Grammatikspezifikation

Der Parser-Generator *SableCC* wurde im Rahmen der Diplomarbeit von [ANS] erweitert. Zu diesem Zweck wurde die Grammatik des SableCC-Parser-Generators ein wenig modifiziert, so dass die neuen Schlüsselwörter und die Typangaben während der Auswertung berücksichtigt werden. Daneben wurde die semantische Analyse entsprechend erweitert.

In dieser Arbeit wurde es notwenig, kleine Modifizierungen an der Grammatikspezifikation und an der semantischen Analyse vorzunehmen. Das hat folgenden Grund: in der abstrakten Syntax - spezifiziert mittels JastAdd - lässt sich der Sternoperator `*` angeben, der für eine beliebige Anzahl von Terminalen/Nichtterminalen steht. JastAdd generiert daraus ein Attribut, das den Typ `List` zugewiesen bekommt. Nun wird dieser Typ ebenfalls von JastAdd generiert und es wird nicht der Typ aus dem Standard Java JDK verwendet. Das bringt ein Problem mit sich. Innerhalb der OCL-Spezifikation muss nun auf den generierten Typ von JastAdd zugeriffen werden. Der Typ `List` kann dabei nicht verwendet werden, da dieser nicht eindeutig ist. SableCC generiert für Ausdrücke mit dem Stern `*` immer eine `java.util.List`, auch wenn die Liste von JastAdd verwendet werden soll. Daher ist es notwendig, einen Typ mit seiner vollständigen Qualifizierung angeben zu können (mit Punktnotation). Dieses Verhalten wurde in der SableCC-Erweiterung nicht implementiert. Aus diesem Grund musste die Spezifikationsdatei für die SableCC-Erweiterung angepasst werden. In dieser wurden die Typangaben so modifiziert, dass nun Punkte zugelassen sind. Die Typangaben werden durch die Regeln

```
ast_type = l_abkt P.external_name r_abkt;

external_name = {simple} T.id
              | {extended} T.external_name
              | ({generic} T.id ) ;
```

in der ursprünglichen Grammatikspezifikation eingeführt. Das Terminal `T.external_name` ist wie folgt definiert:

```
Helpers
...
external_name_start = [ uppercase + lowercase ] ;
external_name_char = [ [ uppercase + lowercase ] + [ '_' + digit ] ] ;

...

Tokens
...
external_name = external_name_start (external_name_char)*;
...
```

Wie zu sehen ist, wird in der Hilfsdefinition von `external_name_char` kein Punkt zugelassen. In der angepassten Spezifikation ist das Terminal `T.external_name` wie folgt spezifiziert:

```
Helpers
...
external_name_start = [ uppercase + lowercase ] ;
external_name_char = [ [ [ uppercase + lowercase ] + [ '_' + digit ] ] + '.' ];
...

Tokens
...
{normal} external_name = external_name_start (external_name_char)*;
...
```

Das führt zu einem neuen Problem. Mit dem Schlüsselwort `Package` soll ein Paketname angegeben werden können. Die semantische Analyse der SableCC-Erweiterung nimmt den Paketnamen als einzelne Bestandteile entgegen, die Typangabe dagegen soll als einzelnes Morphem gelten. Durch die Hinzufügung des Punktes zum Terminal `external_name` werden alle Zeichenketten, die einen Punkt und Buchstaben enthalten als dieses Terminal erkannt. Somit wird bspw. der Paketname `superpackage.parser` nicht als die Morphemfolge `pkg_id`, `dot`, `pkg_id` erkannt, sondern als `external_name`, da die Verarbeitung nach dem Prinzip der längsten Kette erfolgt.

In der ursprünglichen SableCC-Spezifikation wurden zu den Morphemdefinitionen die zwei Zustände `normal` und `package` hinzugefügt. Sobald das Schlüsselwort `Package` gelesen wird, wird der Zustand `package` aktiv, solange bis ein Semikolon gelesen wird. Um das Terminal `external_name` auszuschließen, wird es in den Zustand `normal` gepackt, damit ist es im Zustand `package` nicht mehr erreichbar.

Anpassung der Klassengenerierung

Mit den im vorherigen Abschnitt gezeigten Änderungen kann der Parser Typnamen erkennen, die einen Punkt enthalten. Die semantische Analyse muss aber ebenfalls ein wenig angepasst werden. In einer SableCC-Spezifikation können Typen verwendet werden, ohne sie vollständig qualifizieren zu müssen.

Damit die semantische Analyse den vollständigen Typnamen ermitteln kann, wird in der Klasse `TypeMap` eine Abbildung von Paket- auf Typnamen definiert. Während der semantischen Analyse wird der nicht vollständig qualifizierte Typname in einen vollständig qualifizierten umgeformt.

Für den OCL-Parser folgt hier ein Ausschnitt aus dieser Klasse:

```
public class TypeMap extends Object {
...
    private Map packagesByTypeName = new HashMap();

    private void initializePackageDirectory() {
        Map m = packagesByTypeName;
        ...
        addPackage(m, "tudresden.ocl20.pivot.ocl2parser.gen.ocl2as",
          new String[]
            {
                "OclFileAS",
                "PackagedConstraintAS",
                "ExpressionInOclAS",
                "ConstraintAS",
                ...
```

Die Methode `addPackage` nimmt eine Hashtabelle, einen Paktnamen und ein String-Array entgegen und fügt alle Einträge des String-Arrays in die Hashtabelle so ein, dass das Element des String-Arrays als Schlüssel und der Paketname immer als Wert eingetragen wird. Der Paketname wird so mit jedem angegebenen Typen des String-Arrays verkünpft. Dieser Paketname wird während der semantischen Analyse dem Typen als Präfix angefügt, so dass ein vollqualifizierter Name entsteht.

Ziel ist es nun, einen bereits voll qualifizierten Typnamen nicht mit Hilfe der Klasse `TypeMap` aufzulösen, da der Name bereits vollständig vorliegt. Die semantische Analyse befindet sich in der Datei `org.sablecc.sablecc.GenAnalyses`. Die Methode `createLAttrEvalAdapter()` ist dabei für die Generierung der Klasse `LAttrEvalAdapter` zuständig. Diese Methode benötigt die Typeinformationen und arbeitet mit der Klasse `TypeMap`. Der geänderte Quelltext wurde durch zwei Kommentarzeilen deutlich gemacht. In Listing 6.1 ist der Quelltext abgebildet.

```
1  ...
2  /* Change by Nils to support full qualified type names in the
3     grammar specification****/
4  // If a dot is part of the type name then take the whole part name,
5  // otherwise get the package name and append the type name
6
7  String astTypeAlternativeQualified = null;
8  if (astTypeAlternative.contains(".")) {
9            astTypeAlternativeQualified = astTypeAlternative;
10 }
11 else {
12           astTypeAlternativeQualified =
13           typemap.getPackageForType(astTypeAlternative) + "." +
                 astTypeAlternative;
14 }
15
16 String astTypeElementQualified = null;
17 if (astTypeElement.contains(".")) {
18           astTypeElementQualified = astTypeElement;
19 }
20 else {
21           astTypeElementQualified =
```

```
22              typemap.getPackageForType(astTypeElement) + "." +
                    astTypeElement;
23  }
24  /****************** Change end ***************/
```

Listing 6.1: geänderter Quelltext zur Erzeugung der Klasse `LAttrEvalAdapter`

6.2.5 Generierung des OCL-Parsers

Die Grammatikspezifikation und die abstrakte Syntax sind nun vollständig. Bevor der Parser mit der SableCC-Erweiterung generiert werden kann, müssen zwei kleinere Eingriffe in den SableCC-Quelltext vorgenommen werden. Diese Anpassungen könnten unnötig gemacht werden, sofern die Grammatik für die SableCC-Spezifikation angepasst werden würde (was aus Zeitmangel während dieser Arbeit leider nicht möglich war), indem die benötigten Informationen in die Grammatikspezifikation aufgenommen werden.

Der erste Eingriff betrifft das Bekanntmachen der Typen der abstrakten Syntax mit SableCC. Dazu wird in der Klasse *org.sablecc.sablecc.TypeMap* eine *HashTable* aufgebaut. In diese HashTable sind alle Typen eingetragen, die innerhalb der Grammatikspezifikation verwendet werden. Hier müssen lediglich die Typen manuell eingetragen werden.

Der zweite Eingriff betrifft die Datei *./org/sablecc/sablecc/analysis.txt*. Diese Datei dient als Schablone für den Quelltextgenerator. Hier findet sich der statische Quelltext für die Klassen

- `Analysis`
- `AnalysisWithReturn`
- `AnalysisAdapter`
- `DepthFirstAdapter`
- `ReversedDepthFirstAdapter`
- `LAttrEvalAdapter`
- `AttrEvalException`

wieder. Da innerhalb des generierten Quelltextes auf die abstrakte Syntax zugegriffen wird, müssen für die Dateien `LAttrEvalAdapter.java` und `AnalysisWithReturn.java` zwei `import`-Anweisungen eingefügt werden, die auf die Klassenhierarchie der abstrakten Syntax zeigen.

Der generierte Quelltext benötigt die Klassen `Heritage` und `NodeFactory` (die nicht generiert werden). `Heritage` benötigt eine Methode namens `copy()`, die eine `Heritage`-Instanz zurückgibt. Die Klasse `NodeFactory` stellt eine Fabrikklasse dar, die die Klassen der abstrakten Syntax instanziieren muss (zur Erinnerung: die zu implementierenden `compute`-Methoden bekommen eine Instanz einer Klasse aus der abstrakten Syntax übergeben (siehe Abschnitt 4.2.4 auf Seite 76)). Diese Klasse muss lediglich eine Methode mit der folgenden Signatur implementieren:

```
static public Object createNode(String name)
```

Um den Aufwand der Erzeugung der Instanz zu reduzieren, wurde auf die *Reflection*-API zurückgegriffen. Damit nimmt die `createNode`-Methode folgende Gestalt an:

```
static public Object createNode(String name) {
        String binaryName = new String();
        if (name.equals("Object")) {
                return null;
        }
        if (name.equals("ocl2as.List")) {
                binaryName = name;
        } else {
                binaryName = "ocl2as." + name;
        }

        Class loadedClass;
        try {
                loadedClass = Class.forName(binaryName);
                return loadedClass.newInstance();
        } catch (ClassNotFoundException e) {
                e.printStackTrace();
        } catch (IllegalAccessException ex) {
                ex.printStackTrace();
        } catch (InstantiationException ex) {
                ex.printStackTrace();
        }
        return null;
}
```

Für das Erzeugen des SableCC-Parsers, der abstrakten Syntax und für das Einweben der Aspekte gibt es eine Steuerungsdatei für das Werkzeug *Ant* [ANT]. Diese Steuerungsdatei findet sich unter *./build.xml* wieder. Diese Datei beinhaltet verschiedene Ziele (im Englischen *targets* genannt). In Tabelle 6.1 sind alle Ziele mit ihrer Bedeutung aufgeführt.

Um den OCL-Parser zu generieren, muss eine bestimmte Reihenfolge für die Ausführung der Ziele eingehalten werden. Zunächst muss das Ziel `jastAdd_without_aspects` ausgeführt werden. Dies startet die Generierung der abstrakten Syntax, ohne die Aspekte einzuweben. Anschließend kann das Ziel `sableCC` aufgerufen werden, das den Parser generiert. Für die Generierung müssen die Klassen der abstrakten Syntax vorhanden sein. Zum Schluss muss das Ziel `jastadd` aufgerufen werden. Dieses generiert noch einmal die abstrakte Syntax, webt aber die Aspekte mit ein.

6.2.6 Transformation konkreter Syntaxbaum in abstrakter Syntaxbaum

Der OCL-Parser wurde generiert und die Klassen der abstrakten Syntax erzeugt. Nun muss die Transformation des konkreten Syntaxbaumes in einen abstrakten (als Instanz der abstrakten Syntax) erfolgen. Diese Transformation wird mit dem von SableCC generierten *Framework* durchgeführt. Die Klasse `LAttrEvalAdapter` enthält dazu abstrakte Methoden, die nur noch implementiert werden müssen (siehe Abschnitt 4.2.1 auf Seite 69). Diese Implementierung findet sich im Paket `ocl2Transformer` in der Klasse `Cs2AsOcl2` wieder.

Ziel	Bedeutung
`build`	Führt zunüchst das Ziel `clean` und anschließend das Ziel `gen` aus.
`gen`	Führt zunächst das Ziel `jastAdd` aus und anschließend das Ziel `sableCC`.
`sableCC`	Generiert den SableCC-Parser mit der Spezifikationsdatei `ocl2.parser` und legt den Parser im Ordner `parserFiles` ab.
`clean`	Löscht alle `class`-Dateien und die Verzeichnisse für den generierten SableCC-Parser und für die Klassen der abstrakten Syntax.
`jastAdd`	Generiert die Klassen für die abstrakte Syntax mittels der Spezifikationsdatei *./src/spec/jastadd/ocl2.ast*. Nach der Generierung werden alle Aspekte unter *./src/spec/jastadd* eingewebt.
`jastAdd_without_aspects`	Generiert nur die Klassen für die abstrakte Syntax, ohne die Aspekte einzuweben.

Tabelle 6.1: Ziele der Ant-Datei und ihre Bedeutung

Prinzipiell gestaltet sich das Schreiben der Methoden sehr einfach, denn die `compute`-Methoden bekommen als Parameter alle Werte übergeben, die nur noch in eine Instanz einer Klasse der abstrakten Syntax verpackt werden müssen.

Als Beispiel dient die folgende Grammatikregel aus der Spezifikation des OCL-Parsers:

```
postfix_exp_cs<OclExpressionAS>= ...
| {arrowright_iterate}<IterateExpAS> postfix_exp_cs
  arrowright iterate open_paren iterate_var_cs?
  initialized_variable_cs verticalbar
  ocl_expression_cs close_paren
...
```

Diese Alternative gibt eine Instanz der Klasse `IterateExpAS` zurück. Diese Klasse ist in der abstrakten Syntax folgendermaßen definiert (alle Superklassen sind ebenfalls dargestellt):

```
abstract OclExpressionAS;

abstract CallExpAS : OclExpressionAS ::= [Source:OclExpressionAS];
abstract LoopExpAS : CallExpAS ::= Body:OclExpressionAS;

IterateExpAS : LoopExpAS ::= [Iterator:VariableAS] Result:VariableAS;
```

Diese Zeilen besitzen folgende Bedeutung: ein `Iterate`-Ausdruck kann einen Quell-(`Source`-)-Ausdruck haben (in diesem Beispiel besitzt er einen). Auf jeden Fall ist ein `Body`-Ausdruck vorhanden, es kann eine `Iterate`-Variable geben, aber es muss eine Ergebnis-(`Result`-)Variable vorhanden sein. Wenn man sich die Zeile der konkreten Syntax ansieht, stellt man fest, dass sie mit der abstrakten Syntax korrespondiert. Die nicht ausgelassenen Grammatiksymbole (wie `postfix_exp_cs`, `iterate_var_cs`, `initialized_variable_cs` und `ocl_expression_cs`) werden nun der `compute`-Methode

übergeben (als Instanzen von Klassen der abstrakten Syntax), deren Signatur im Folgenden dargestellt ist:

```
public IterateExpAS computeAstFor_AArrowrightIteratePostfixExpCs(
                IterateExpAS myAst,
                Heritage nodeHrtg,
                OclExpressionAS astPostfixExpCs,
                VariableAS astIterateVarCs,
                VariableAS astInitializedVariableCs,
                OclExpressionAS astOclExpressionCs
) throws AttrEvalException
```

Die Methode bekommt schon eine `IterateExpAS`-Instanz übergeben, deren Attribute nur noch mit den anderen Parameter gesetzt werden müssen (siehe Listing 6.2).

```
1 myAst.setSource(astPostfixExpCs);
2 myAst.setBody(astOclExpressionCs);
3
4 if (astIterateVarCs != null) myAst.setIterator(astIterateVarCs);
5 myAst.setResult(astInitializedVariableCs);
6 return myAst;
```

Listing 6.2: Implementierung der `compute`-Methode für *IterateExpAS*-Instanz

In Zeile 4 wird eine Überprüfung der Variable `astIterateVarCs` durchgeführt. Dies ist notwendig, da der Nutzer keine Iterator-Variable angegeben haben muss. In der abstrakten Syntax ist die Iterator-Variable optional. Daher darf diese nicht gesetzt werden, wenn sie nicht vorhanden ist (siehe auch Abschnitt 5.1.3 auf Seite 90).

Neben diesem einfachen Zusammensetzen liefert die Transformation in einen abstrakten Syntaxbaum ein wenig mehr. In den `compute`-Methoden

- `computeAstFor_ADotOperationCallPostfixExpCs`
- `computeAstFor_AArrowRightOperationCallPostfixExpCs`
- `computeAstFor_AParameterPropertyCallExpCs`

werden die zusätzlichen Attribute `ArrowRightExpression` und `DotExpression` der Klasse `OperationCallExpAS` gesetzt, abhängig davon, ob die Operation als Kollektions- oder normale Operation eingegeben wurde.

Im Abschnitt 6.2.1 auf Seite 103 wurde die folgende Regel angeführt:

```
operation_constraint_cs = ...
                        | {empty} op_constraint_stereotype_cs colon;
```

Solch eine leere Bedingung soll in den abstrakten Syntaxbaum nicht übernommen werden. Daher liefert die zugehörige `compute`-Methode dieser Regel nur `null` und keine Instanz einer Subklasse von `OperationConstraintAS` zurück. In der Methode `computeAstFor_AOperationContextDeclarationCs` wird dieser Rückgabewert ausgewertet. Wenn alle Bedingungen des OCL-Ausdrucks leer sind (die `post`, `inv`, `post`, ... Bedingungen besitzen keinen Ausdruck), wird eine Ausnahme generiert, andernfalls werden alle nicht leeren Bedingungen zu einem `OperationConstraintAS`-Objekt zusammengesetzt.

Da die Nachrichten (im Englischen *Messages*) vom Pivotmodell nicht unterstützt werden, in der Grammatik aber spezifiziert sind, wird in den entsprechenden `compute`-Methoden eine Ausnahme generiert, mit dem Hinweis, dass diese Ausdrücke nicht implementiert wurden.

Die Methode `computeAstFor_AIterateVarCs` bekommt keine Instanz einer Klasse der abstrakten Syntax übergeben (siehe Schlüsselwort `#nocreate` in der Grammatikspezifikation, Nichtterminal `iterate_var_cs`). Die Methode gibt eine Instanz der Klasse `VariableAS` zurück, aber das Nichtterminal `formal_parameter_cs` liefert eine Instanz der Klasse `VariableExpAS` zurück. Da eine `VariableExpAS`- auf eine `VariableAS`-Instanz zeigt, wird einfach diese Instanz zurückgegeben.

Neben den `compute`- und `createNode`-Methoden gibt es noch drei Methoden, die Quelltext kapseln, der an vielen Stellen gebraucht wird. Die Methode `fillTokenAS` nimmt eine Instanz der Klasse `Token` (generiert von SableCC) entgegen und erzeugt aus dieser eine `TokenAS`-Instanz der abstrakten Syntax. Die Methode `fillOperationCallAS` dient dazu arithmetische Ausdrücke in eine Instanz der Klasse `OperationCallExpAS` zu transformieren. Die Methode `transformListVariableExp` letztendlich transformiert eine Liste von `VariableExpAS`-Instanzen in eine Liste von `VariableAS`-Instanzen.

6.2.7 Einbinden des OCL-Parsers in das Dresdner OCL-Toolkit

In der Arbeit [Braeuer] wurde das gesamte *Toolkit* in Eclipse-PlugIn's aufgeteilt. Darunter befindet sich auch das PlugIn `tudresden.ocl20.pivot.parser`, das Schnittstelleninformationen für einen Parser zur Verfügung stellt. Hier wird eine Schnittstelle `IOclParser` definiert, die jeder Parser implementieren muss. Für die Schnittstelle gibt es aber schon eine Basisimplementierung in Form der abstrakten Klasse `AbstractOclParser`, so dass der Implementierungsaufwand auf ein paar wenige Methoden sinkt. Diese Klasse muss nur noch abgeleitet werden, um den Parser zu implementieren. Das Parser-PlugIn stellt über die Methode `getParser` in der Klasse `ParserPlugin` eine Methode zur Verfügung, mit dem Aufrufer eine Parser-Instanz (Schnittstelle `IOclParser`) bekommen können. Diese Methode wurde für den OCL-Parser, der in dieser Arbeit entsteht, angepasst.

6.3 Semantische Prüfungen

Im vorhergehenden Abschnitt wurde gezeigt, wie der eigentliche OCL-Parser generiert und welche Klassenstruktur verwendet wird, um abstrakte Syntaxbäume darzustellen. Nachdem die Transformation des konkreten Syntaxbaumes in einen abstrakten Syntaxbaum abgeschlossen ist, muss eine semantische Prüfung durchgeführt werden, denn nicht jeder OCL-Ausdruck, der syntaktisch korrekt ist, ergibt einen Sinn. Das folgende Beispiel zeigt einen syntaktisch korrekten OCL-Ausdruck:

```
context Person
    inv: age > 0
```

Ergibt dieser Ausdruck Sinn? Dazu muss man wissen, ob es im betrachtenden Modell eine Klasse `Person` (bzw. *Type* im Pivotmodell) mit dem Attribut `age` (bzw. *Property*) gibt. Wenn dem so ist, ergibt der Ausdruck einen Sinn. Wenn die Klasse `Person` oder das Attribut `age` kein Bestandteil des Modells sind, ist der Ausdruck nicht korrekt.

Neben der semantische Prüfung muss der abstrakte Syntaxbaum in einen abstrakten Syntaxgraphen transformiert werden. Der abstrakte Syntaxgraph ist eine Instanz von

EssentialOCL (siehe Pakte `tudresden.ocl20.pivot.essentialocl`). Beides - sowohl die semantische Analyse als auch die Transformation in den abstrakten Syntaxgraph - wird in einer rekursiven Berechnung durchgeführt. Das heißt, jeder Klasse der abstrakten Syntax (generiert von JastAdd) erhält eine Methode namens

```
computeASM(Environment env)
```

Diese Methode startet abhängig von der Besitzerklasse einen rekursiven Aufruf zu den Kindelementen. Diese rekursive Traversierung stoppt bei den Blättern (Instanzen der Klasse `TokenAS`). Während der Traversierung werden die semantischen Regeln geprüft. Jede `computeASM`-Methode liefert als Ergebnis eine Instanz einer Klasse aus EssentialOCL wieder. Die erste `computeASM`-Methode, die aufgerufen wird, befindet sich in der Klasse `OclFileAS`. Diese Methode liefert eine Liste von Namensräumen zurück. Jeder Namensraum enthält die Bedingungen, die der Nutzer spezifiziert hat (siehe Aspekt *OclFileASMComputation*). Namensräume werden durch die Schnittstelle `Namespace` repräsentiert. Diese Schnittstelle ist Bestandteil des Pivotmodells (siehe Paket `tudresden.ocl20.pivot.pivotmodel`).

Da die semantischen Prüfungen über alle Klassen der abstrakten Syntax verstreut sind, bietet es sich an, diese Methoden mittels Aspektdateien hineinzuweben. So können Methoden, die zusammengehören, aber dennoch über Klassen verstreut sind, zentralisiert werden, was das Verständnis vereinfacht. JastAdd übernimmt diese Arbeit (siehe Abschnitt 5.1.5 auf Seite 92). Die Aspekte sind in `jadd`-Dateien im Verzeichnis *./src/spec/jastadd* zu finden.

Der rekursiven Berechnung des abstrakten Syntaxgraphen liegt das Interpretermuster nach [GOF] zugrunde.

Dieser Abschnitt beschäftigt sich zunächst mit der Umgebung, die während der Auswertung des abstrakten Syntaxbaumes genutzt wird. Anschließend werden alle Aspekte, die die semantische Analyse und die Berechnung des abstrakten Syntaxgraphen durchführen, beschrieben.

6.3.1 Die Klasse Environment

Während der semantischen Analyse wird die Klasse `Environment` benötigt. Sie nimmt implizite und explizite Variable auf (siehe beispielsweise Nichtterminal `IteratorExpCS` in [OCL2] auf Seite 73 und Nichtterminal `LetExpCS` in [OCL2] auf Seite 87) und speichert den aktuellen Namensraum. Darüberhinaus wird sie für die Speicherung von Kontextinformationen und für die Suche von Modellelementen verwendet.

In [OCL2] auf Seite 91 bis 93 ist eine Umgebung mit unterschiedlichen Operationen spezifiziert. Es hat sich im Laufe der Implementierung herausgestellt, dass ein paar dieser Operationen nicht gebraucht werden und dass manch neue Operation hinzugefügt werden musste. Im Anhang C sind beide Spezifizierungen gegenübergestellt.

6.3.2 Der Aspekt AtPreResolver

In OCL-Ausdrücken kann der Postfix `@pre` in einer post-Bedingung, die sich auf eine Methode/Operation bezieht, gesetzt werden. Der Parser lässt den Postfix aber auch in allen anderen Bedingungen zu. Dies könnte durch eine Anpassung der Grammatik vermieden werden, allerdings würde diese dann sehr umfangreich werden, da ein vollständig eigener Regelsatz um das Nichtterminal `ocl_expression_cs` (siehe SableCC-Spezifikation) gebildet werden müsste. Um diesen Aufwand zu sparen, wurde der Aspekt *AtPreResolver*

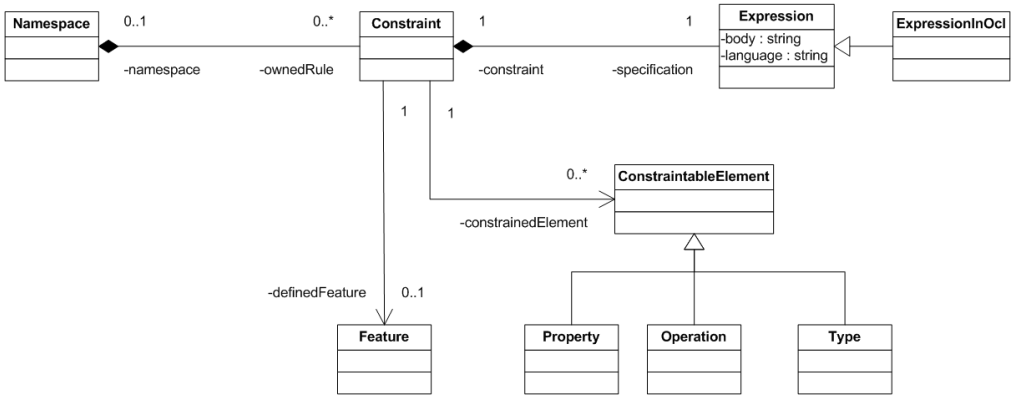


Abbildung 6.2: Klassenstruktur für das Ergebnis des Parsers

geschrieben. Dieser prüft, ob der Postfix `@pre` tatsächlich ausschließlich in `post`-Bedingungen verwendet wurde. Zu diesem Zweck erhält jede Klasse der abstrakten Syntax eine Methode namens `resolveAtPre`. Diese Methoden durchlaufen den abstrakten Syntaxbaum mittels eines rekursiven Abstiegs. Falls der Postfix `@pre` an einer falschen Stelle verwendet wurde, wird eine `AtPreException` generiert, andernfalls wird keine Rückmeldung gegeben.

Der abstrakte Syntaxbaum wird von oben nach unten durchlaufen. Oben steht die Information, welche Bedingungsart (`body`, `post`, `pre`, `inv`, ...) einem OCL-Ausdruck zugeordnet werden soll. Weiter unten im Baum kann der Postfix `@pre` als Knoten vorkommen. Das heißt, die Information, um welche Bedingungsart es sich handelt, muss nach unten durchgereicht werden. Dazu wird die Klasse `AtPreEnvironment` eingeführt. Diese besitzt genau zwei Attribute: `predecessor` und `isPostConstraint`. Das Attribut `predecessor` speichert die Vorgängerumgebung. Das Attribut `isPostConstraint` ist vom Typ `boolean` und wird immer dann auf `true` gesetzt, wenn eine `post`-Bedingung vorliegt. Letzteres muss etwas konkretisiert werden. Die Klasse `OperationConstraintAS` besitzt das "Attribut" `OperationStereoType` (siehe abstrakte Syntax), was die Bedingungsart trägt. Anhand dieser Information wird das Attribut `isPostConstraint` gesetzt. Wenn im weiteren Verlauf der Traversierung ein `@pre`-Knoten gefunden wird und das Attribut `isPostConstraint` nicht gesetzt ist, wird eine `AtPreException` generiert.

6.3.3 Der Aspekt BodyStringComputation

Das Ergebnis des gesamten Parser-Vorgangs ist ein abstrakter Syntaxgraph, der Instanz von EssentialOCL ist. Der Parser liefert nach der Berechnung eine Liste von `Namespace`-Instanzen aus dem Pivotmodell zurück (das Pivotmodell befindet sich im Paket `tudresden.ocl20.pivot.pivotmodel`). Eine `Namespace`-Instanz besitzt mehrere `Constraint`-Instanzen (siehe Paket `tudresden.ocl20.pivot.essentialocl`). Eine `Constraint`-Instanz verweist auf eine `Expression`-Instanz und besitzt darüberhinaus auch ein Attribut namens `body` vom Typ `String`. Zusammengefasst sind die Klassenbeziehungen in Abbildung 6.3.3.

Das Attribut `body` speichert den OCL-Ausdruck für die Bedingung als Zeichenkette. Da diese Zeichenkette vom generierten Parser nicht zurückgeliefert wird, muss diese neu aus dem abstrakten Syntaxbaum zusammengesetzt werden. Genau dieses berechnet

der *BodyStringComputation*-Aspekt. Dazu wird in jeder Klasse der abstrakten Syntax eine Methode namens `computeBodyString()` eingewebt, die immer den Rückgabetyp `String` besitzt. Es wird wiederum ein rekursiver Abstieg durch den abstrakten Syntaxbaum durchgeführt, wobei die Zeichenketten nach und nach zusammengesetzt werden.

6.3.4 Der Aspekt AbstractComputeMethods

Dieser Aspekt dient lediglich dazu, abstrakte Methoden in abstrakte Klassen einzuweben. Die folgenden Klassen sind betroffen:

- `LiteralExpAS`
- `OclExpressionAS`
- `CallExpAS`
- `FeatureCallExpAS`
- `LoopExpAS`

Eine Methode namens `transformString2List`, die eine Repräsentation einer Paketstruktur (beispielsweise `A::B::C`) in eine Liste von einzelnen Elementen zerlegt (ohne die Zeichen `::`), wird in die Klasse `FeatureCallExpAS` eingewebt.

6.3.5 Der Aspekt CollectionLiteralExpASMComputation

Dieser Aspekt dient zur Berechnung von Kollektionsliteralen, beispielsweise gehören dazu:

- `Set{1,2,3}`
- `Bag{1,2,3}`
- `Sequence{1,2,3}`
- `Set{1..3}`

Zudem berechnet der Aspekt die Werte, die der Kollektion übergeben werden, das heißt, es werden Instanzen der Subklassen von `CollectionItem` und `CollectionRange` berechnet.

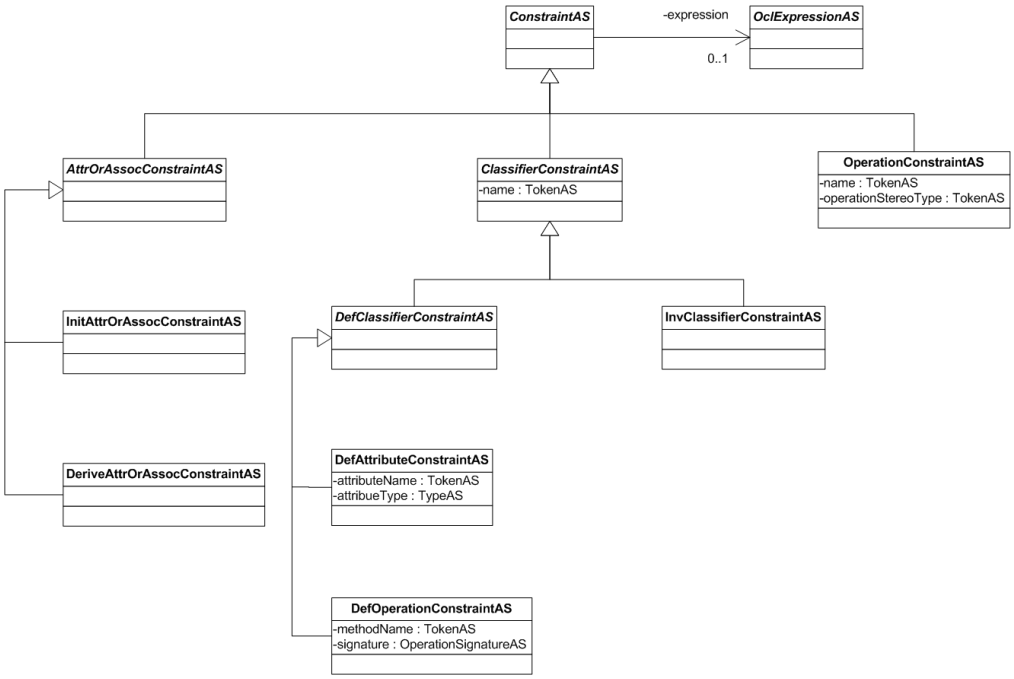


Abbildung 6.3: Klassenhierarchie der Bedingungen in der abstrakten Syntax

In der Klassenhierarchie der abstrakten Syntax sind folgende Klassen von dem Aspekt betroffen:

- `CollectionLiteralExpAS`
- `CollectionLiteralPartAS`
- `CollectionItemAS`
- `CollectionRangeAS`

6.3.6 Der Aspekt ConstraintASMComputation

Dieser Aspekt berechnet Instanzen der Klasse `Constraint` (zugehörig zum Pivotmodell). In der abstrakten Syntax gibt es eine kleine Klassenhierarchie der Bedingungen, die in Abbildung 6.3 dargestellt ist.

Alle Bedingungsarten werden in diesem Aspekt berechnet. Hier wird auch eine Instanz der Klasse `ExpressionInOcl` (aus EssentialOCL) erzeugt (siehe Abbildung 6.3.3). Der Aufbau von EssentialOCL und der abstrakten Syntax unterscheiden sich in den Bedingungen. Die invariante Bedingung, die Definitionen von Operationen und Attributen werden in der abstrakten Syntax indirekt durch den Klassennamen gespeichert. Beispielsweise gibt es für die invariante Bedingung die Klasse `InvClassifierConstraintAS`. In EssentialOCL dagegen wird die Bedingungsart in einer Instanz vom Typ `Constraint` als Attribut (siehe Aufzählung `ConstraintKind`) gespeichert.

6.3.7 Der Aspekt ContextASMComputation

Dieser Aspekt stößt die Berechnung der Bedingungen an. Gegeben sei folgenden OCL-Ausdruck:

```
context A
        inv: b < 0
        inv: c > 50
```

Dieser Ausdruck besitzt zwei invariante Bedingungen, die sich auf die Klasse `A` (bzw. *Type* im Pivotmodell) beziehen. In der abstrakten Syntax gibt es zwar eine Klasse `ContextAS`, die Instanzen der Klasse `ConstraintAS` halten, aber nicht in EssentialOCL. Im Pivotmodell gibt es eine Klasse `Constraint`, die zwei wesentliche Referenzen hält (siehe Abbildung 6.3.3). Die eine Referenz zeigt auf das Element im Modell (eine Instanz aus der Klassenhierarchie des Pivotmodells), das mit der Bedingung verknüpft ist. In der konkreten Syntax wird dieses Element nach dem Schlüsselwort `context` geschrieben (im Beispiel würde die `Constraint`-Instanz eine Referenz auf das Element `A` aus dem Modell halten). Die zweite Referenz der Klasse `Constraint` zeigt auf eine Instanz der Klasse `Expression` (aus dem Pivotmodell). `Expression` ist lediglich eine Superklasse, deren Ableitungen Bedingungen in unterschiedlichen Sprachen darstellen. In diesem Fall gibt es nur eine Unterklasse mit Namen `ExpressionInOcl` aus EssentialOCL. Diese Klasse kapselt die eigentliche Bedingung. An der `Constraint`-Instanz hängt immer nur genau eine `ExpressionInOcl`-Instanz, während im konkreten Ausdruck mehrere Bedingungen zu einem Kontext zusammengefasst werden. Das bedeutet, es muss für jede Bedingung ein eigenes `Constraint`- und `ExpressionInOcl`-Objekt erzeugt werden. Daher ergibt sich eine Liste von `Constraint`-Objekten als Rückgabewert der `computeASM`-Methode in diesem Aspekt. In Abbildung 6.4 sind beide Darstellungen aufgeführt. Oben befindet sich die Instanz der abstrakten Syntax und unten die Instanz von EssentialOCL.

Innerhalb der Berechnungsmethoden werden die beiden impliziten Variablen `self` und `result` (sofern es sich um eine `post`-Operationsbedingung handelt) erzeugt und zur weiteren Verarbeitung der Umgebung hinzugefügt.

6.3.8 Der Aspekt ErrorTokenComputation

Falls während der semantischen Analyse ein Fehler auftritt, muss dem Benutzer angezeigt werden, wo sich der Fehler im OCL-Ausdruck befindet. Zu diesem Zweck bilden `TokenAS`-Instanzen die Blätter des abstrakten Syntaxbaumes. Wenn ein semantischer Fehler kurz oberhalb eines Blattes auftritt, kann die Information des Morphems zur Fehlerausgabe genutzt werden. Nicht so, wenn der Fehler an einem inneren Knoten aufgedeckt wird. Dazu ein Beispiel. Gegeben sei folgender OCL-Ausdruck:

```
context Person::age
inv : '18'
```

Eine invariante Bedingung muss immer einen booleschen Rückgabewert liefern. Die ist hier nicht der Fall. Die Überprüfung, ob der Typ des `inv`-Ausdruckes ein boolescher Wert ist, kann erst entschieden werden, wenn der Ausdruck berechnet wurde, das heißt, wenn der entsprechende Teilbaum abgearbeitet wurde (in diesem Fall das Literal `'18'`). In diesem Fall wird die Überprüfung also innerhalb des Baumes durchgeführt. Da kein Blattknoten "in der Nähe" ist (das Schlüsselwort `inv` wird ausgelassen), wird die Ausgabe einer Fehlernachricht mit Positionsangabe schwierig. An dieser Stelle setzt der `ErrorTokenComputation`-Aspekt an. Er durchläuft rekursiv einen Teilbaum und liefert

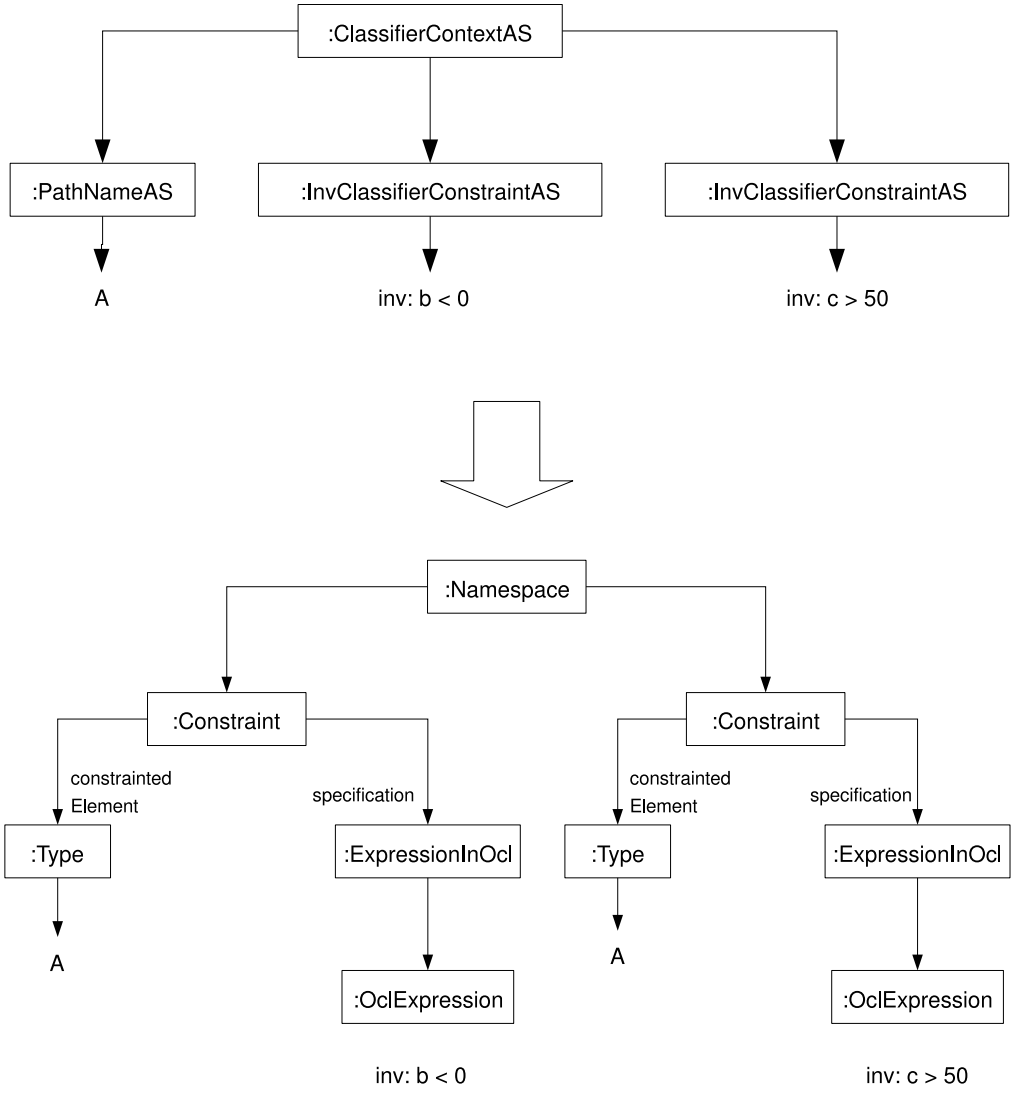


Abbildung 6.4: Transformation einer *ContextAS*-Instanz in eine Instanz von Essential-OCL

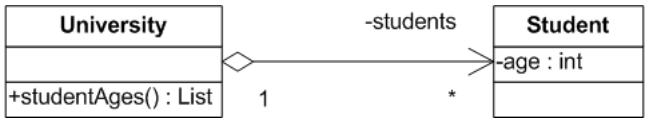


Abbildung 6.5: Beispiel-Modell für implizite Iterator-Ausdrücke

die erste `TokenAS`-Instanz zurück, die er findet. Diese Instanz wird dann verwendet, um aus ihr die Positionsangabe zu beziehen. Dadurch kann es passieren, dass die Fehlerangabe nicht ganz genau wird.

Ein Fehlermorphem wird nur für Instanzen der Subklassen von `OclExpressionAS` berechnet.

6.3.9 Der Aspekt IfExpASMComputation

Dieser Aspekt berechnet einen `IF`-Ausdruck (in EssentialOCL `IfExp`). Dazu werden die Bedingung und die Zweige `then` und `else` berechnet.

6.3.10 Der Aspekt IterateExpASMComputation

Dieser Aspekt dient zur Berechnung der `IterateExp`-Instanzen. Er ist sehr einfach aufgebaut, so dass er hier nicht weiter betrachtet wird.

6.3.11 Der Aspekt IteratorExpASMComputation

Der `IteratorExpASMComputation`-Aspekt berechnet Iteratoren-Ausdrücke, die sich auf eine Operation oder Attribut (bzw. *Property* im Pivotmodell) beziehen. Ein Iterator-Ausdruck kann explizit oder implizit auftreten. Explizit wird er durch den Pfeiloperator `->`. Dazu ein Beispiel. Gegeben sei folgenden OCL-Ausdruck:

```
context ProgramPartner
inv: self.programs.partners -> select(p:ProgramPartner | p<>self)
```

Der Teilausdruck `self.programs.partners` bezeichnet einen Kollektionstyp (nur auf diesen können Iteratorenoperationen angewandt werden). (Hinweis: Nach einem Pfeil kann auch eine normale Kollektionsoperation wie `union` stehen. Solch ein Ausdruck wird nicht in eine `IteratorExp`-Instanz transformiert, sondern in eine `OperationCallExp`-Instanz.) Die Grammatik "erkennt" einen expliziten Ausdruck, so dass die Behandlung dessen zu einem "normalen" Zusammenbauen führt. Es sei darauf hingewiesen, dass eine implizite Iterator-Variable erzeugt wird, sofern vom Benutzer keine angegeben wurde (siehe [OCL2], Seite 73, Abschnitt *Synthesized attributes*, Fall [A]). Für das Berechnen der expliziten Iterator-Ausdrücke wird die Methode `computeASM` in die Klasse `IteratorExpAS` eingewebt.

Implizite Iteratoren sind schwieriger zu behandeln. Sie treten immer dann auf, wenn ein Attribut (*Property*) oder eine Operation nach einem Punktoperator steht und der Ausdruck vor dem Punktoperator ein Kollektionstyp ist. Dazu ein Beispiel. Es sei das Modell in Abbildung 6.5 gegeben.

Für dieses Modell soll folgender OCL-Ausdruck gelten:

```
context University:studentAges()
body: students.age
```

Der Teilausdruck `students` ist ein Kollektionstyp aufgrund der *-Assoziation zur Klasse `Student`. Was bedeutet es, wenn ein Attribut (*Property*) auf einen Kollektionstypen mit dem Punktoperator folgt? Der Ausdruck, der sich auf das Attribut `age` bezieht, wird folgendermaßen transformiert:

`students.age` ⇒ `students->collect(age)`

Diese Transformation wird im `IteratorExpASMComputation`-Aspekt definiert, für Attribute (*Property*) und für Operationen. Dazu wird die Methode `computeIteratorASM` in die Klassen `PropertyCallExpAS` und `OperationCallExpAS` eingewebt. Diese Methode wird gegebenenfalls in der `computeASM`-Methoden der beiden Klassen aufgerufen. Dazu muss es aber notwendig sein, dass der Quellausdruck einen Kollektionstypen entspricht (siehe Abschnitte zu den Aspekten `PropertyCallExpASMComputation` und `OperationCallExpASMComputation`).

6.3.12 Der Aspekt LetExpASMComputation

Dieser Aspekt berechnet die Instanz `LetExp` aus EssentialOCL, wobei dieser in die Klasse `LetExpAS` eingewebt wird. In diesem Aspekt werden die einzigen expliziten Variablen berechnet und der Umgebung mittels der Methode `addVariable` hinzugefügt.

6.3.13 Der Aspekt NamespaceASMComputation

Innerhalb eines OCL-Quelltextes kann mit dem Schlüsselwort `package` der Namensraum festgelegt werden, auf den sich alle Suchvorgänge von Klassen (bzw. *Type* im Pivotmodell) beziehen.

Dieser Aspekt webt in die Klasse `PackagedConstraintAS` eine `computeASM`-Methode ein. Diese Methode sucht den Namensraum, der mit dem Schlüsselwort `package` verknüpft ist, berechnet die Bedingungen (Abstieg in den abstrakten Syntaxbaum) und fügt diese dem Namensraum hinzu. Zudem wird der Namensraum in der Umgebung gesetzt. Zuletzt wird dieser Namensraum zurückgegeben.

6.3.14 Der Aspekt OclFileASMComputation

An der Wurzel des abstrakten Syntaxbaumes steht eine Instanz der Klasse `OclFileAS`. Auch in diese Wurzelklasse wird mittels des Aspektes `OclFileASMComputation` die Methode `computeASM` hineingewebt. Diese Methode stößt die gesamte semantische Analyse und die Berechnung des abstrakten Syntaxgraphen an. Der Rückgabewert ist eine Liste von Namensräumen. Die Namensräume werden während der Berechnung gesucht (die Namensräume sind Bestandteil des Modells) und an ihnen werden die Bedingungen hinzugefügt.

6.3.15 Der Aspekt OperationCallExpASMComputation

Mit Hilfe dieses Aspektes werden die Operationsaufrufe in einem OCL-Ausdruck ausgewertet. Da es viele verschiedene Fälle gibt, ist dieser Aspekt einer der umfangreichsten in der gesamten Implementierung.

Der Aspekt webt die Methoden `computeASM`, `computeOperationASM` und zwei Hilfsmethoden in die Klasse `OperationCallExpAS` ein. Wie im Abschnitt 6.3.11 beschrieben, wird auch die Methode `computeIteratorASM` in die Klasse eingewebt.

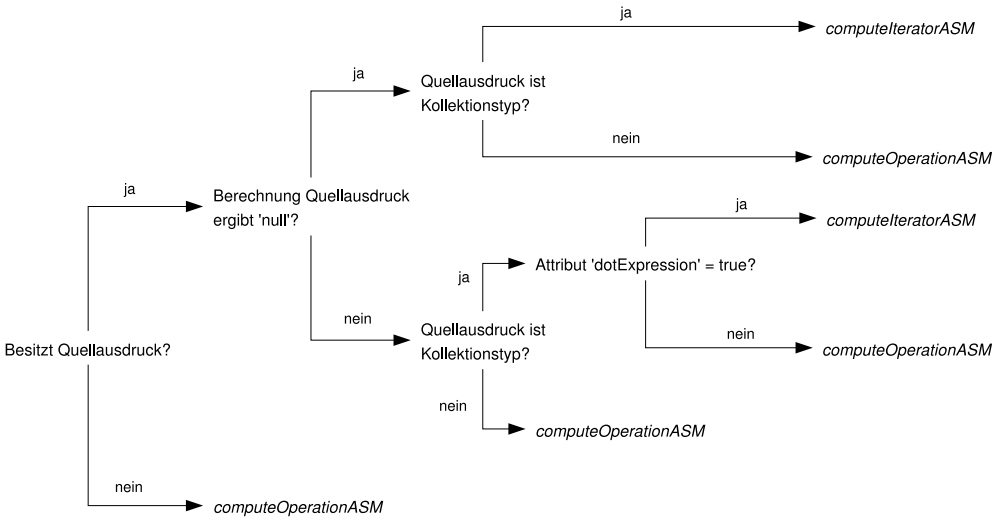


Abbildung 6.6: Entscheidungsdiagramm *OperationCallExpASMComputation*-Aspekt, Methode *computeASM*

Bei einem Operationsaufruf sind zunächst zwei Fälle zu unterscheiden. Es kann sich um einen normalen Operationsaufruf handeln, der deklarierte Methoden innerhalb einer Klasse (bzw. *Type* im Pivotmodell) aufruft, oder eine implizite Kollektionsoperation (siehe Abschnitt 6.3.11). Wenn es sich um einen normalen Operationsaufruf handelt, kann ein Quellausdruck fehlen wie im folgenden Beispiel:

```
context Person
post: eat() >= 3
```

Die Methode `eat()` wird ohne Quellausdruck angegeben. Aus diesem Grund muss eine implizite Variable gesucht werden, deren Klasse diese Methode besitzt. Vorausgesetzt die Klasse (bzw. *Type*) `Person` verfügt über eine Methode `eat()`, wird die implizite Variable `self` benutzt. So wird der vorhergehende OCL-Ausdruck in folgenden transformiert:

```
context
post: self.eat() >= 3
```

Das entspricht der Grammatikregel [D] und [F] des Nichtterminals `OperationCallExpCS` in [OCL2] auf Seite 80. In der Methode `computeASM` wird die Entscheidung getroffen, ob die Methode `computeOperationASM` oder `computeIteratorASM` aufgerufen wird. Das Ergebnis der Entscheidung ist abhängig davon, ob der Operationsaufruf über einen Quellausdruck verfügt oder nicht und wenn ja, ob dieser Quellausdruck einen Kollektionstypen entspricht. In Abbildung 6.6 ist ein Entscheidungsdiagramm für die Methode `computeASM` gegeben. Aus diesem geht hervor, unter welchen Bedingungen die Methode `computeOperationASM` oder `computeIteratorASM` aufgerufen werden. Wenn die Entscheidung zugunsten der Methode `computeOperationASM` getroffen wird, müssen viele Fälle unterschieden werden, die unterschiedliche Ergebnisse zur Folge haben. Diese Fälle sind in Abbildung 6.7 dargestellt.

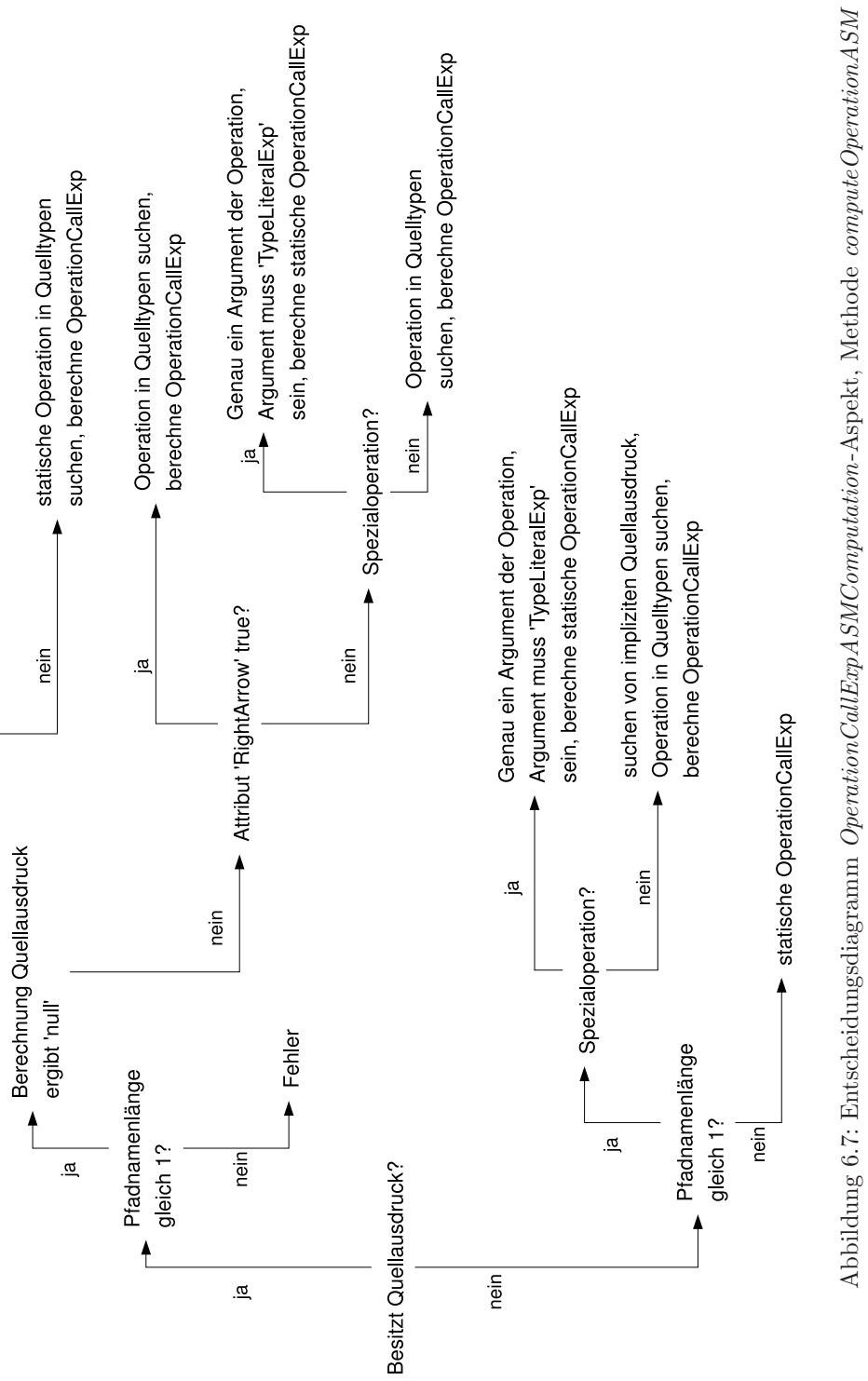


Abbildung 6.7: Entscheidungsdiagramm *OperationCallExpASMComputation*-Aspekt, Methode *computeOperationASM*

Es gibt in diesem Aspekt eine Besonderheit. Die Umgebung besitzt das Attribut `specialOclOperation` vom Typ `boolean`. Dieses wird in diesem Aspekt gesetzt, wenn festgestellt wird, dass es sich um eine der Spezialoperationen `oclIsKinfOf`, `oclIsTypeOf` oder `oclAsType` handelt. Damit wird dem rekursiven Kindaufruf, der die Parameter der Operation berechnet, der Hinweis gegeben, dass es sich bei diesem Parameter um einen Typen handeln muss. Im Aspekt `PropertyCallExpASMComputation` wird diese Information verwendet.

Im Quelltext sind die einzelnen Fälle ausführlich beschrieben, so dass an dieser Stelle nicht weiter darauf eingegangen wird.

6.3.16 Der Aspekt OperationSignatureASMComputation

Dieser Aspekt webt in die Klasse `OperationSignatureAS` die Berechnungsmethode `computeASM` hinzu, die die Parameter und gegebenenfalls den Rückgabewert einer Operation berechnet.

Da die Signatur in der Spezifikation der abstrakten Syntax von ihrer Operation losgelöst ist, musste ein zusätzlicher Typ für diesen Rückgabewert geschaffen werden. Dazu wurde die Klasse `OperationSignatureContainer` eingeführt, die nicht Bestandteil von EssentialOCL ist. Dieser Typ nimmt lediglich die berechneten Werte auf, um sie weiter zu reichen (vgl. Klasse `BlockContainerAS` in Abschnitt 3.1 auf Seite 52).

6.3.17 Der Aspekt PackageAspect

Alle Aspekte werden in die Klassen der abstrakten Syntax hineingewebt. Der Quelltext der Aspekte benötigt oft fremde Typen wie beispielsweise die Elemente aus dem Pivotmodell oder aus EssentialOCL. Das bedeutet, die benötigen Klassen müssen importiert werden. Alle Importanweisungen befinden sich zentral in diesem Aspekt.

6.3.18 Der Aspekt PrimivitveLiteralExpASMComputation

In diesem Aspekt werden alle primitiven Literale behandelt (diese sind Subklassen von `PrimitiveLiteralExp` aus EssentialOCL). Die Erzeugung der Literalausdrücke ist sehr einfach, so dass an dieser Stelle nicht näher darauf eingegangen wird.

Eine Besonderheit gibt es bei den Zeichenkettenliteralen zu beachten. Die Zeichenkettenliterale werden vom Lexer identifiziert und an den Parser weitergegeben, allerdings ist das Literal durch zwei Apostrophe eingeschlossen. Diese Darstellung befindet sich auch im abstrakten Syntaxbaum. Die beiden Apostrophe werden entfernt, bevor sie in den abstrakten Syntaxgraphen aufgenommen werden.

6.3.19 Der Aspekt PropertyCallExpASMComputation

Dieser Aspekt berechnet alle Attributausdrücke (bzw. *Property*-Ausdrücke im Pivotmodell). Die Berechnung ist ähnlich komplex wie die Analyse der Operationsausdrücke (siehe Abschnitt 6.3.15). In die Klasse `PropetyCallExpAS` werden von diesem Aspekt drei Methoden eingewebt:

- `computeASM`
- `computePropertyASM`
- `computeTypeLiteralExpression`

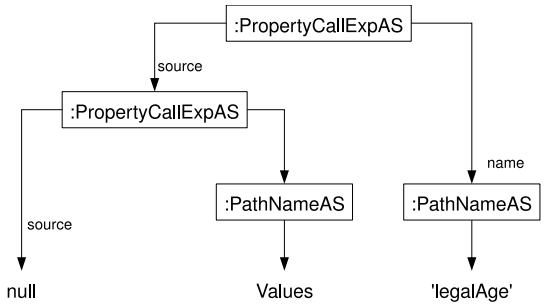


Abbildung 6.8: Objektgraph für Ausdruck *Values.legalAge*

Das Vorgehen ist ähnlich wie bei den Operationsausdrücken. Die `computeASM`-Methode überprüft zunächst, ob es sich bei dem Attribut (*Property*) um einen Ausdruck mit oder ohne Quellausdruck handelt. Wenn ein Quellausdruck vorhanden ist, muss geprüft werden, ob dieser einem Kollektionstypen entspricht. (Zur Erinnerung: Der Aspekt `IteratorExpASMComputation` webt die Methode `computeIteratorASM` ein.) Wenn es sich um ein normales Attribut (*Property*) handelt, das heißt, es gehört nicht zu einer Instanz von `IteratorExp`, muss geprüft werden, ob die Berechnung von einer Spezialoperation (`oclIsKindOf`, ...) angestoßen wurde. Dies wird über das Abfragen des Attributes `specialOclOperation` der Umgebung erreicht.

Eine Instanz der Klasse `PropertyCallExpAS` kann auch auf eine Klasse (*Type*) zeigen. Dazu folgendes Beispiel:

```
context Person
inv: age >= Values.legalAge
```

Betrachtet wird der Teilausdruck `Values.legalAge`. Dieser setzt sich aus der Klasse (*Type*) `Values` und dem statischen Attribut (*Property*) `legalAge` zusammen. Für den Ausdruck `Values.legalAge` ist der Ast des abstrakten Syntaxbaumes in Abbildung 6.8 dargestellt.

Die zweite `PropertyCallExpAS`-Instanz, an der der Pfadname mit dem Wert `Values` hängt, besitzt keinen Quellausdruck. Eine `PropertyCallExp`-Instanz soll aus dem gesamten Teilbaum gebaut werden, die auf das statische Attribut (*Property*) `legalAge` zeigt. Dazu muss in der Klasse (*Type*) `Values` nach dem Attribut `legalAge` gesucht werden. Das geschieht wie folgt: Die Berechnung wertet erst die zweite `PropertyCallExpAS`-Instanz aus. Deren `computeASM`-Methode soll nun eine `PropertyCallExp`-Instanz zurückgeben, obwohl keine `PropertyCallExp`-Instanz gebaut werden kann, da `Values` eine Klasse ist. Trotzdem muss `Values` im Modell gesucht und zurückgegeben werden, damit bei der Auswertung von `legalAge` in der Klasse `Values` gesucht werden kann. Um an diesen Typen zu kommen, wurde der Umgebung das Attribut `soureType` mitgegeben. Dieses Attribut wird während der Auswertung einer `PropertyCallExpAS`-Instanz gesetzt, sofern erkannt wird, dass diese auf eine Klasse (*Type*) zeigt. Der Rückgabewert der `computeASM`-Methode ist in diesem Fall `null`, um anzuzeigen, dass in der Umgebung das Attribut `sourceType` gesetzt wurde. Während der Auswertung der darüberliegenden `PropertyCallExpAS`-Instanz (siehe Beispiel) kann diese Information verarbeitet werden. Bei Rückgabe von `null`, wird nach der Klasse (*Type*) in der Umgebung gesucht und anschließend wird in dieser Klasse (*Type*) nach dem Attribut (*Property*), auf das

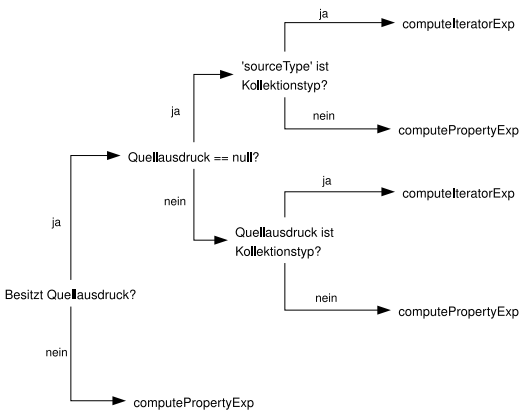


Abbildung 6.9: Entscheidungsdiagramm *PropertyCallExpASMComputation*-Aspekt, Methode *computeASM*

der Pfadname zeigt (im Beispiel `legalAge`), gesucht. Anschließend muss noch geprüft werden, ob das gefundene Attribut (*Property*) statisch ist.

Eine `PropertyCallExpAS`-Instanz kann auch auf eine Aufzählung (im Englischen *Enumeration*) zeigen, wie im folgenden Beispiel:

```
context CostumerCard
inv: color = Color::silver
```

Hier bezeichnet `Color::silver` das Aufzählungsliteral. Dieser Ausdruck wird durch eine einzelne `PropertyCallExpAS`-Instanz repräsentiert. Ein Aufzählungstyp kann an der Länge eines Pfadnamens erkannt werden. Er enthält mehr als ein Element. Darüberhinaus kann auch ein Pfadname mit mehreren Elementen eine Klasse (*Type*) bezeichnen, wie im folgenden Beispiel:

```
context Person
inv: age >= Package1::Values.legalAge
```

Die Auswertung einer `PropertyCallExpAS`-Instanz erfolgt in zwei Schritten (vgl. Klasse `OperationCallExpAS`): zuerst muss entschieden werden, welche Berechnungsmethode aufgerufen werden soll: `computePropertyASM` oder `computeIteratorASM`. Diese Entscheidung wird in der Methode `computeASM` getroffen. In Abbildung 6.9 zeigt ein Entscheidungsdiagramm, unter welchen Bedingungen die weiteren Berechnungen erfolgen.

Wenn die Methode `computePropertyASM` abgearbeitet wird, müssen viele Fälle unterschieden werden, die unterschiedliche Ergebnisse haben können. Im Entscheidungsdiagramm der Abbildung 6.10 sind alle Fälle mit ihren möglichen Ausgängen dargestellt.

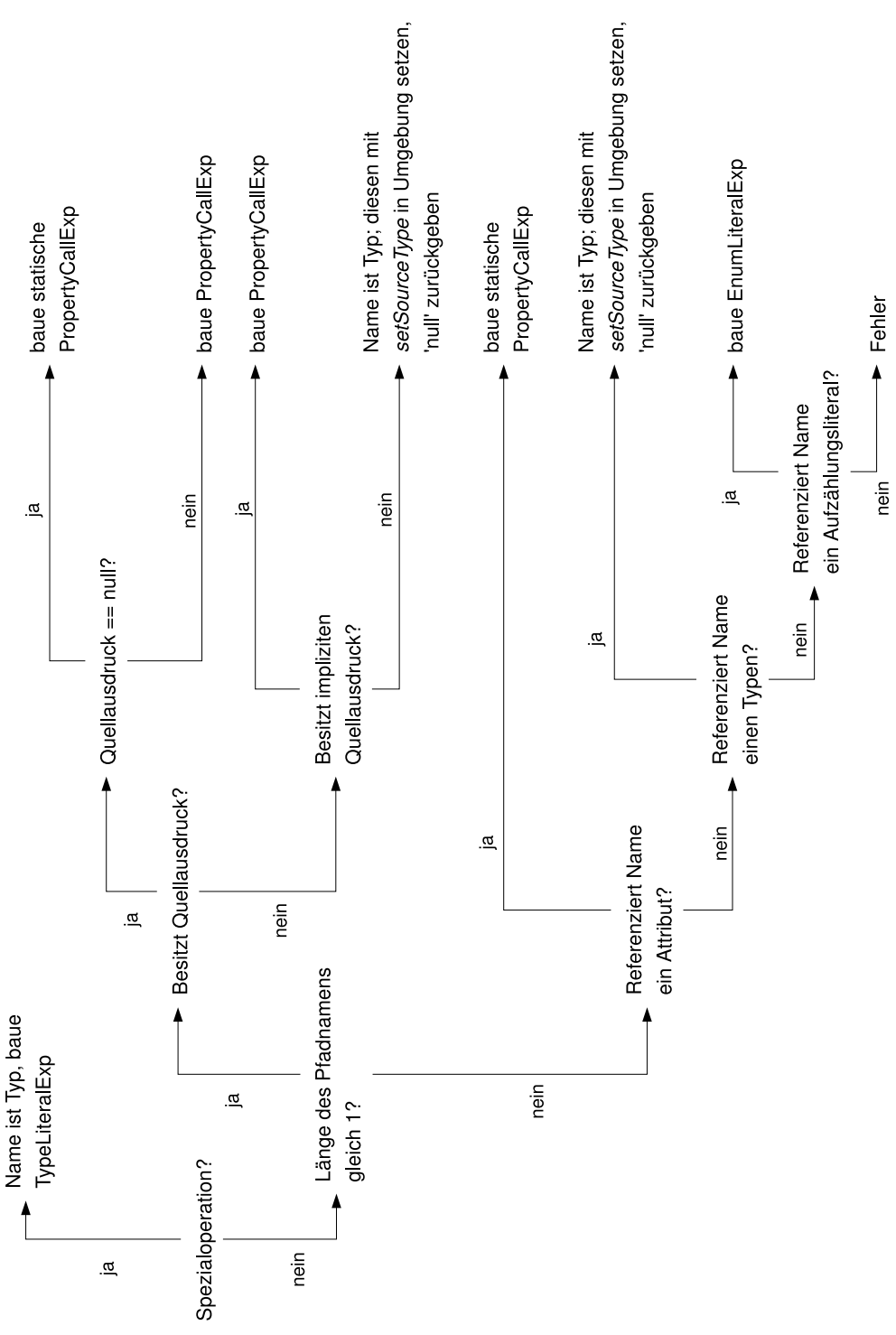


Abbildung 6.10: Entscheidungsdiagramm *PropertyCallExpASMComputation*-Aspekt, Methode *computePropertyASM*

6.3.20 Der Aspekt TupleLiteralExpASMComputation

Dieser Aspekt wertet die Tupelliterale der Form

```
tuple { elemente }
```

aus, wobei `elemente` für eine Liste von weiteren Literalen steht. Der Aspekt webt zwei `computeASM`-Methoden ein: in die Klasse `TupleLiteralExpAS` und `TupleLiteralPartAS`.

6.3.21 Der Aspekt TypeASMComputation

In diesem Aspekt werden Typausdrücke berechnet. Einige Typausdrücke sind im folgenden dargestellt:

```
Set(Integer)
TupleType(name:String, age:Integer)
```

Dazu werden den Klasse `PathNameAS`, `CollectionTypeAS` und `TupleTypeAS` jeweils eine `computeASM`-Methode eingewebt.

6.3.22 Der Aspekt VariableASMComputation

Dieser Aspekt fasst die Berechnung von `Variable`- und `VariableExp`-Instanzen zusammen, indem in die Klassen `VariableAS` und `VariableExpAS` die `computeASM`-Methoden eingewebt werden.

7 Abschließende Betrachtung

In diesem Kapitel soll eine abschließende Betrachtung zum OCL-Parser gegeben werden. Zunächst wird eine kurze Zusammenfassung des Beleges gegeben. Anschließend wird der Parser einer Bewertung unterzogen und am Ende wird auf die Erweiterbarkeit des Parsers eingegangen.

7.1 Zusammenfassung

Aufgabe dieses Beleg war es, den alten OCL-Parser des Dresdner OCL-Toolkits an die Pivotschnittstelle anzupassen und ihn zu vervollständigen. Für diese Aufgabe war es notwendig, sich intensiv mit dem Thema Parserbau zu beschäftigen (siehe Kapitel 2 auf Seite 19). Dabei fiel auf, dass es Änderungen im klassischen Parserbau gegeben hat. Während im klassischen Parserbau mit Symboltabellen gearbeitet wurde, gibt es heute Möglichkeiten, diese einzusparen und statt dessen mit Objektgraphen zu arbeiten. Mit dem Werkzeug *JastAdd* erfuhr dieser Beleg eine gute Unterstützung und auch die Ideen in [JAST] trugen zu dem neuen OCL-Parser bei.

Ein weiterer Schwerpunkt dieser Arbeit bildete die Überarbeitung der Grammatik aus [ANS] für die Arbeit mit einer abstrakten Syntax. Dabei musste auch die konkrete Syntax aus [OCL2] betrachtet werden, was zunächst sehr verwirrend war, denn diese Grammatik ist für den Bau eines Parsers nicht geeignet; sie gibt eher intuitiv vor, wie OCL-Ausdrücke zusammengesetzt werden können.

Nachdem die OCL-Grammatik einen stabilen Zustand erreicht hatte, konnte mit der Implementierung des OCL-Parsers begonnen werden. Die größte Schwierigkeit betraf die Umsetzung der semantischen Regeln mittels Aspekte, genau die Umsetzung der Operations- und Attributaufrufe, da diese sehr umfangreich sind und viele Spezialfälle enthalten (siehe Abschnitt 6.3.15 auf Seite 121 oder auch Abschnitt 6.3.19 auf Seite 124).

Die Integration des Parser in das Dresdner OCL-Toolkit konnte recht einfach bewerkstelligt werden, da im Rahmen von [Braeuer] eine Schnittstelle bzw. eine abstrakte Klasse für die Anbindung zur Verfügung gestellt wurde.

7.2 Bewertung des Parsers

Der alte Parser wurde ausschließlich unter Verwendung der SableCC-Erweiterung implementiert. Die Auswertung der semantischen Regeln erfolgt innerhalb der Übersetzung des konkreten Syntaxbaumes in den abstrakten Syntaxgraph (der Instanz der abstrakten Syntax von OCL ist). Dabei entfällt die Übersetzung in einen abstrakten Syntaxbaum. Der Parser ist somit vollständig an die SableCC-Erweiterung gekoppelt. Des Weiteren wird die Erzeugung des abstrakten Syntaxgraphen, einschließlich der Auswertung der semantischen Regeln, in einer einzigen Datei vorgenommen, was die Einarbeitung und Wartung erschwert.

Der neue OCL-Parser erzeugt einen abstrakten Syntaxgraphen (als Instanz von EssentialOCL) in zwei Phasen (siehe Abschnitt 6.1.2 auf Seite 100). Die erste Phase wird dabei

mit der SableCC-Erweiterung realisiert. Sie übersetzt einen Quelltext zunächst in einen konkreten Syntaxbaum, um diesen in einen abstrakten Syntaxbaum zu transformieren. Der notwendige Quelltext für die letztgenannte Transformation hält sich in Grenzen. Ein Austausch der SableCC-Erweiterung ist einfach umzusetzen, wenn der neue Parser-Generator die Bedingung erfüllt, dass er dem Nutzer eigenen Quelltext einfügen lässt, um so die Klassen der abstrakten Syntax instanziieren zu können. Zu dem muss der generierte Parser ein paar kleine Besonderheiten implementieren (siehe Abschnitt 6.2.1 auf Seite 103). Dadurch wird der Austausch des Parser-Generators etwas erschwert.

Die zweite Phase der Übersetzung wird mit dem Werkzeug *JastAdd* unterstützt. JastAdd generiert aus einer Beschreibung die Klassen für die abstrakte Syntax. Theoretisch könnten diese auch per Hand oder durch ein anderes Werkzeug erzeugt werden. Wichtig sind die von JastAdd generierten Methoden, die die Aspekte während der semantischen Analyse benutzen. Die Aspektdateien könnten auch von Werkzeugen wie AspectJ [AspectJ] in die Klassen der abstrakten Syntax hineingewebt werden. Dabei müssen die Aspektdateien gegebenenfalls an die Syntax des neuen Werkzeugs angepasst werden. Ein Austausch von JastAdd wäre mit Anpassungen an andere Werkzeuge möglich.

Durch die Aufteilung der Aufgaben auf zwei Werkzeuge ist es möglich, andere Werkzeuge einzusetzen, so dass ein hoher Grad an Flexibilität ermöglicht wird.

Die semantische Analyse bezieht sich auf das Pivotmodell, somit wird eine Abhängigkeit hergestellt, die die Anpassung an andere Metamodelle schwierig werden lässt. Da das Pivotmodell geschaffen wurde, um eine hohe Flexibilität im Umgang mit Metamodellen zu erreichen, wiegt die Abhängigkeit weniger schwer.

Ein Ziel dieser Arbeit war es, den OCL-Parser zu vervollständigen. Da das Pivotmodell nicht alle Konstrukte aus UML unterstützt, musste die Verarbeitung von Nachrichten-Ausdrücken aufgegeben werden. In der Grammatik werden diese aber spezifiziert. Das Pivotmodell unterstützt zur Zeit keine Assoziationsklassen, womit OCL-Ausdrücke, die sich auf solche beziehen, zu Fehlermeldungen führen würden.

Jede Software sollte ausgiebig getestet werden. In dieser Arbeit wurde bisher lediglich getestet, ob der Parser korrekte OCL-Ausdrücke verarbeitet, das heißt, keine Fehlermeldung ausgibt. Die OCL-Ausdrücke, die diesen Test bilden, stammen aus [War]. Dieses Buch zeigt an einem UML-Modell die Verwendung unterschiedlicher OCL-Ausdrücke. Dabei werden die meisten Möglichkeiten der OCL abgedeckt. Aus diesem Grund schien es naheliegend, diese Ausdrücke zu verwenden, da sie einen Großteil des Sprachumfangs beschreiben. Mit diesem bisher sehr einfachen Test konnte nur geprüft werden, ob der Parser die OCL-Ausdrücke akzeptiert. Es konnte jedoch nicht getestet werden, ob der Parser die richtigen Instanzen der EssentialOCL zurückgibt. Zu diesem Zweck wurde während dieser Arbeit begonnen, einen Testfallgenerator zu implementieren. Dieser generiert anhand einer Beschreibung *JUnit*-Testfälle ([JUnit]), die nur ausgeführt werden müssen. Die Beschreibung der Testfälle geschieht dabei in eine für diesen Zweck geschaffenen DSL (*domain specific language*). In dieser wird der zu prüfende OCL-Ausdruck zusammen mit dem zu erwartenden Ergebnis eingegeben. Idee dieser Vorgehensweise ist es, mit möglichst vielen dieser Beschreibungen, den Parser gründlich zu testen. Falls ein Fehler gemeldet wird, kann der OCL-Ausdruck, der zu dem Fehler führt, in eine neue Beschreibungsdatei aufgenommen werden. Anschließend wird der Fehler beseitigt und man kann alle Testfälle neu generieren und ausführen, um so die Zuverlässigkeit des Parsers zu prüfen.

7.3 Ausblick

Software entwickelt sich im Laufe der Zeit weiter, da sich die Anforderungen ändern. So könnte die OCL-Spezifikation in Zukunft teilweise gravierend geändert werden, so dass eine Anpassung des OCL-Parser notwendig wird. Änderungen können sich dabei auf drei Bereiche der Spezifikation beziehen. Diese werden hier kurz angesprochen:

- Änderung einer semantischen Regel
  - Anpassung der zugehörigen Aspektdatei
- Änderung der konkreten Syntax
  - Grammatikspezifikation des Parser-Generators anpassen
  - Rückgabetyp der neuen/angepassten Regel bestimmen
  - gegebenenfalls einen neuen Typen in der abstrakten Syntax einführen
  - Transformation konkrete Syntax zu abstrakten Syntaxbaum anpassen
  - einen neuen Aspekt erzeugen oder bestehenden Aspekt ändern
- Änderung der abstrakten Syntax
  - Änderung im Paket `tudresden.ocl20.pivot.essentialocl` vollziehen
  - abstrakte Syntax in *ast*-Datei (JastAdd) anpassen
  - Typen der Grammatikregeln innerhalb der Parser-Generator-Spezifikation müssen angepasst werden
  - Einfügen neuer Aspekte / Änderung bestehender Aspekte

Zu beachten ist, dass immer geprüft werden muss, ob das Pivotmodell die Änderung unterstützt. Gegebenenfalls muss dieses geeignet angepasst werden.

Bisher konnte der Parser noch nicht zusammen mit dem in [BRA] entstandenen Interpreter getestet werden. Dies müsste in späteren Arbeiten nachgeholt werden.

Der OCL-Parser verbindet das in [Braeuer] geschaffene Pivotmodell mit dem in [BRA] geschaffenen Interpreter. Damit erreicht das Dresdner OCL-Toolkit eine neue Version. Die Module aus der vorhergehenden Version müssten noch an das Pivotmodell angepasst werden, was Raum für weitere Arbeiten lässt.

A Attributierungsregeln der Sprache PL0

A.1 Variablen

In der folgenden Auflistung sind alle semantischen Regeln, die die Variablen betreffen, aufgeführt.

- Regel:VARDECL → 'var' ident ';'
  - ererbtes Attribut: -
  - synthetisiertes Attribut:

    ```
    NameableAS element = VARDECL.env.lookupLocal(ident.value);
    if (element == null) {
    element = VARDECL.env.lookup(ident.value);
    if (element instanceOf ProcedureAS) -> Fehler
    VariableAS var = new VariableAS(ident.value);
    VARDECL.env.addVariable(var);
    List<VariableAS> variableList = new List();
    variableList.add(var);
    VARDECL.ast = variableList;
    }
    -> Fehler
    ```
  - Beschreibung: Vor den Variablen können Konstanten deklariert werden. Aus diesem Grund wird erst geprüft, ob es einen Identifizierer mit dem Namen `ident.value` in der lokalen Umgebung gibt. Falls ja, kann es sich nur um eine Konstante handeln, und führt zu einem Fehler, weil Konstanten und Variablen in einem Blockniveau unterschiedliche Namen haben müssen. Nun kann eine Prozedur denselben Namen haben wie die deklarierte Variable. Aus diesem Grund wird in allen Umgebungen nachgesehen (`lookup`). Wenn es ein Element gibt, was eine Prozedur ist, dann führt dies zu einem Fehler. Andernfalls kann es kein Element mit diesem Namen geben (was in Ordnung ist) oder es kann bereits eine Variable oder eine Konstante mit demselben Namen in einem höheren Blockniveau geben, dann wird diese überdeckt. Zum Schluss wird eine Liste von Variablen erzeugt und dieser die aktuelle Variable hinzugefügt. Diese Liste wird dem Nichtterminal `VARDECL` übergeben (siehe Tabelle 3.7).

- Regel: VARDECL → 'var' ident VARENUM ';'
  - ererbtes Attribut: `VARENUM.env = VARDECL.env`
  - synthetisiertes Attribut:

    ```
    NameableAS element = VADECL.env.lookupLocal(ident);
    if (element == null) {
    ```

```
element = VARDECL.env.lookup(ident);
if (element instanceOf ProcedureAS) -> Fehler
List<VariableAS> variableList = new List();
VariableAS var = new VariableAS(ident.value);
variableList.add(var);
VARENUM.env.addVariable(var);
variableList.addAll(VARNEUM.ast);
VARDECL.ast = variableList;
}
-> Fehler
```

  - Beschreibung: Die Umgebung wird an das Nichtterminal `VARENUM` weitergegeben, weil die Variable `var` auch dort zugänglich gemacht werden muss. Erst wird geprüft, ob der Identifizierer `var.value` in der lokalen Umgebung vorhanden ist. Falls ja, kann es sich nur um eine vorher deklarierte Konstante mit demselben Namen handeln. Dies führt zu einem Fehler, da Variablen und Konstanten getrennte Namen haben müssen. Anschließend wird geprüft, ob der Identifizierer in allen Umgebungen vorkommt. Insbesondere kann es sein, dass eine Prozedur mit demselben Namen deklariert wurde. Falls dies zutrifft, führt dies zu einem Fehler, da die deklarierte Variable nicht denselben Namen tragen darf wie eine vorher deklarierte Prozedur. Eine Konstante oder Variable in einem höherem Blockniveau mit demselben Namen kann es durchaus geben, dann wird diese von der aktuellen Variablen überdeckt. Zum Schluss wird eine Liste von Variablen angelegt, die die aktuelle Variable aufnimmt (diese muss erst noch mit dem Namen `var.value` erzeugt werden). Natürlich müssen auch alle weiteren Variablen hinzugefügt werden, die das Nichtterminal `VARENUM` mitbringen (in Form einer Liste (siehe dazu die Tabelle 3.7)). Abschließend wird die Variableliste dem Nichtterminal `VARDECL` übergeben.

- Regel: VARENUM → ',' ident
  - ererbtes Attribut: -
  - synthetisiertes Attribut:

```
NameableAS element = VARENUM.env.lookupLocal(ident.value);
if (element == null) {
element = VARENUM.env.lookup(ident.value);
if (element instanceOf ProcedureAS) -> Fehler
List<VariableAS> variableList = new List();
VariableAS var = new VariableAS(ident.value);
VARENUM.env.addVariable(var);
variableList.add(var);
VARENUM.ast = variableList;
}
-> Fehler
```

  - Beschreibung: Das Nichtterminal `VARENUM` steht für die Aufzählung von mehreren Variablen. Das bedeutet, dass Variablen schon vorher deklariert wurden. So muss geprüft werden, ob die Variable mit dem Namen `ident.value` schon vorhanden ist. Aus diesem Grund wird in der lokalen Umgebung nachgesehen. Als Ergebnis kann eine Variable zurückgegeben werden oder eine Konstante. Wenn es ein Element mit demselben Namen gibt, dann führt

dies zu einem Fehler, weil Konstanten und Variablen im selben Blockniveau nicht den gleichen Namen tragen dürfen. Falls es kein Element mit dem Namen gibt, wird in allen Umgebungen nachgesehen. Es könnte sein, dass es eine Prozedur mit demselben Namen gibt. Dann führt dies zu einem Fehler. Falls diese beiden Prüfungen erfolgreich absolviert wurden, wird eine Liste von Variablen angelegt, die die aktuelle Variable hinzugefügt bekommt (diese muss erst neu angelegt werden). Zum Schluss wird dem Nichtterminal `VARENUM` diese Liste hinzugefügt.

- Regel: VARENUM1 → ',' ident VARENUM2
  - ererbtes Attribut: `VARENUM2.env = VARENUM1.env`
  - synthetisiertes Attribut:

```
NameableAS element = VARENUM1.env.lookupLocal(ident.value);
if (element == null) {
element = VARENUM1.env.lookup(ident.value);
if (element instanceOf ProcedureAS) -> Fehler
List<VariableAS> variableList = new List();
VariableAS var = new VariableAS(ident.value);
variableList.add(var);
VARENUM2.env.addVariable(var);
variableList.addAll(VARNEUM2.ast);
VARENUM1.ast = variableList;
}
-> Fehler
```

  - Beschreibung: Die neue Variable `ident` steht mitten in der Variablendeklaration. Es muss geprüft werden, ob diese Variable in der aktuellen Umgebung schon deklariert wurde. Falls ja, führt dies zu einem Fehler (siehe vorhergehende Regel). Andernfalls muss geprüft werden, ob dieser Identifizierer in allen Umgebungen vorkommt. Wenn dies so ist und es sich dabei um eine Prozedur handelt, führt dies zu einem Fehler. Andernfalls wird eine Liste von Variablen erzeugt und die aktuelle Variable hinzugefügt (diese muss noch erzeugt werden). Die Umgebung wird an das zweite Nichtterminal `VARNEUM2` weitergereicht. Die aktuelle Variable wird aber hinzugefügt. Abschließend werden die Variablen des Nichtterminals `VARNEUM2` der aktuellen Variablenliste hinzugefügt und dem Nichtterminal `VARENUM1` zugewiesen.

- Regel: STATEMENT → ident ':=' EXPRESSION
  - ererbtes Attribut: `EXPRESSION.env = STATEMENT.env`
  - synthetisiertes Element:

```
NameableAS element = STATEMENT.env.lookup(ident.value);
if (element == null) -> Fehler
if (element instanceOf VariableAS) {
AsignStatementAS stmt = new AssignStatementAS(element,
EXPRESSION.ast);
STATEMENT.ast = stmt;
}
-> Fehler
```

  - Beschreibung: Der Identifizierer auf der linken Seite einer Zuweisung kann nur einer Variablen entsprechen, da man weder Konstanten noch Prozeduren

Werte zuweisen kann. Aus diesem Grund wird mittels der Umgebung nach diesem Identifizierer gesucht. Falls dieser nicht gefunden wird, wurde die Variable nicht deklariert, was zu einem Fehler führt. Auch kann es passieren, dass ein Element gefunden wird, es sich aber um eine Konstante oder eine Prozedur handelt, was ebenfalls zu einem Fehler führt. Andernfalls wird ein neuer Zuweisungsbefehl (`AssignStatementAS`) erzeugt und die gefundene Variable mit dem Wert des Nichtterminals `EXPRESSION.ast` verknüpft. Abschließend erhält das Nichtterminal `STATEMENT` diesen Wert.

- Regel: STATEMENT → '?' ident
  - ererbtes Attribut: -
  - synthetisiertes Attribut:

    ```
    NameableAS element = STATEMENT.env.lookup(ident.value);
    if (element == null) -> Fehler
    if (element instanceOf VariableAS) {
    InputStatementAS stmt = new InputStatementAS(element);
    STATEMENT.ast = stmt;
    }
    -> Fehler
    ```

  - Beschreibung: Der Eingabebefehl kann sich nur auf eine Variable beziehen, da nur in dieser die Eingabe gespeichert werden kann. Aus diesem Grund wird in der Umgebung nach einer Variable mit diesem Wert gesucht (`ident.value`). Wenn es keine Variable mit diesem Namen gibt, dann wurde diese nicht deklariert oder es handelt sich um keine Variable. Wenn die Prüfung erfolgreich verlief, wird ein Eingabebefehl mit dieser Variablen erzeugt und am Ende dem Nichtterminal `STATEMENT` hinzugefügt.

- Regel: FACTOR → ident
  - ererbtes Attribut: -
  - synthetisiertes Attribut:

    ```
    NameableAS element = FACTOR.env.lookup(ident.value);
    if (element == null) -> Fehler
    if (element instanceOf ProcedureAS) -> Fehler
    ExpressionAS exp = element;
    FACTOR.ast = exp;
    ```

  - Beschreibung: Das Nichtterminal `FACTOR` steht für einen Ausdruck. Es dient in der Grammatik lediglich dazu, die Prioritäten der mathematischen Operatoren zu sichern. Ein Ausdruck kann entweder eine Zahl, eine Konstante oder eine Variable enthalten, nicht aber eine Prozedur. Aus diesem Grund wird auf die Prozedur geprüft. Falls es sich um eine Prozedur handelt, führt dies zu einem Fehler. Andernfalls wird das gefundene Element dem Nichtterminal `FACTOR` übergeben. Zur Erinnerung: die Klassen `VariableAS` und `ConstantAS` sind Unterklassen von `ExpressionAS`. Da in diesem Fall nur Konstanten und Variablen in Fragen kommen, ist dies eine gültige Zuweisung.

A.2 Konstanten

In der folgenden Auflistung sind alle semantischen Regeln, die die Konstanten betreffen, aufgeführt.

- Regel: CONSTDECL → 'const' ident '=' number ';'
  - ererbtes Attribut: -
  - synthetisiertes Attribut:

    ```
    NameableAS element = CONSTDECL.env.lookup(ident.value);
    if (element instanceOf ProcedureAS) -> Fehler
    ConstantAS const = new ConstantAS(ident.value, number.value);
    List<ConstantAS> constList = new List();
    constList.add(const);
    CONSTDECL.env.addConstant(const);
    CONSTDECL.ast = constList;
    ```

  - Beschreibung: Hier wird die erste Konstante eines neuen Blocks definiert. Aus diesem Grund kann sofort in allen Umgebungen nach dem Identifizierer nachgesehen werden, ohne über die lokale Umgebung zu gehen (diese ist in diesem Fall leer). Anschließend wird geprüft, ob es sich bei dem gefundenen Element um eine Prozedur handelt. Falls ja, handelt es sich um einen Fehler. Andernfalls wird eine neue Liste von Konstanten erzeugt (siehe Tabelle 3.7). Die deklarierte Konstante wird der Liste hinzugefügt, nachdem sie erzeugt wurde, auch der Umgebung, um für weitere Prüfungen zur Verfügung zu stehen. Abschließend wird die Liste dem Nichtterminal `CONSTDECL` zugewiesen.

- Regel: CONSTDECL → 'const' ident '=' number CONSTASSIGN ';'
  - ererbtes Attribut: `CONSTASSIGN.env = CONSTDECL.env`
  - synthetisiertes Attribut:

    ```
    NameableAS element = CONSTDECL.env.lookup(ident.value);
    if (element instanceOf ProcedureAS) -> Fehler
    ConstantAS const = new ConstantAS(ident.value, number.value);
    CONSTDECL.env.addConstant(const);
    List<ConstantAS> constList = new List();
    constList.add(const);
    constList.addAll(CONSASSIGN.ast);
    CONSTDECL.ast = constList;
    ```

  - Beschreibung: Hier werden mehrere Konstanten deklariert. Der Identifizierer, der hier gebraucht wird, wird in allen Umgebungen gesucht. Falls eine Prozedur mit diesem Namen bereits existiert, führt dies zu einem Fehler. Andernfalls wird eine Liste von Konstanten erzeugt, die die aktuelle Konstante aufnimmt. Auch werden die Konstanten des Nichtterminals `CONSTASSIGN` hinzugefügt (siehe Tabelle 3.7). Am Ende wird die Liste dem Nichtterminal `CONSTDECL` übergeben. Die Konstante muss natürlich der Umgebung hinzugefügt werden, damit sie für weitere Prüfung zur Verfügung steht.

- Regel: CONSTASSIGN → ',' ident '=' number
  - ererbtes Attribut: -
  - synthetisiertes Attribut:

```
NameableAS element = CONSTASSIGN.env.lookupLocal(ident.value);
if (element instanceOf ConstantAS) -> Fehler
element = CONSTASSIGN.env.lookup(ident.value);
if (element instanceOf ProcedureAS) -> Fehler
ConstantAS const = new ConstantAS(ident.value, number.value);
List<ConstantAS> constList = new List();
constList.add(const);
CONSTASSIGN.env.addConstant(const);
CONSTASSIGN.ast = constList;
```

  - Beschreibung: Hier wird die letzte Konstante einer Konstantendeklaration definiert. Es muss geprüft werden, ob diese in der lokalen Umgebung schon einmal vorkommt, da andere Konstanten bereits deklariert wurden. Falls es keine Konstante mit demselben Namen gibt, muss in allen Umgebungen geprüft werden, ob es bereits eine Prozedur mit demselben Namen gibt. Falls ja, führt dies zu einem Fehler. Andernfalls wird eine Liste von Konstanten erzeugt, die die aktuelle Konstante aufnimmt. Diese Liste wird dem Nichtterminal `CONSTASSIGN` zugewiesen (siehe Tabelle 3.7). Auch muss die aktuelle Konstante der Umgebung hinzugefügt werden, damit sie für weitere Prüfungen zur Verfügung steht.

- Regel: CONSTASSIGN1 → ',' ident '=' number CONSTASSIGN2
  - ererbtes Attribut: `CONSTASSIGN2.env = CONSTASSIGN1.env`
  - synthetisiertes Attribut:

```
NameableAS element = CONSTASSIGN1.env.lookupLocal(ident.value);
if (element instanceOf ConstantAS) -> Fehler
element = CONSTASSIGN1.env.lookup(ident.value);
if (element instanceOf ProcedureAS) -> Fehler
ConstantAS const = new ConstantAS(ident.value, number.value);
List<ConstantAS> constList = new List();
constList.add(const);
CONSTASSIGN2.env.addConstant(const);
constList.add(CONSTASSIGN2.ast);
CONSTASSIGN1.ast = constList;
```

  - Beschreibung: Hier wird eine Konstante mitten in einer Reihe von Konstantendeklarationen definiert. Es muss geprüft werden, ob eine Konstante mit dem Namen `ident.value` bereits existiert. Falls ja, führt dies zu einem Fehler. Andernfalls muss in allen Umgebungen geprüft werden, ob eine Prozedur mit diesem Namen existiert, was in diesem Fall wiederum zu einem Fehler führen würde. Falls die beiden Prüfungen erfolgreich verlaufen, wird eine Liste von Konstanten erzeugt und die aktuelle Konstante wird dieser hinzugefügt. Die Konstante muss natürlich in die Umgebung aufgenommen werden, damit sie für die weiteren Prüfungen zur Verfügung steht. Anschließend werden die Konstanten des Nichtterminals `CONSTASSIGN2` der Liste hinzugefügt. Abschließend wird diese Liste dem Nichtterminal `CONSTASSIGN1` zugewiesen.

A.3 Prozeduren

In der folgenden Auflistung sind alle semantischen Regeln, die die Prozeduren betreffen, aufgeführt.

- Regel: PROCEDUREDECL → 'procedure' ident ';' BLOCK ';'
  - ererbtes Attribut: `BLOCK.env = PROCEDUREDECL.env`
  - synthetisiertes Attribut:

```
NameableAS element = PROCEDUREDECL.env.lookupLocal(ident.value);
if (element != null) -> Fehler
BLOCK.env.addProcedure(proc);
ProcedureAS proc = new ProcedureAS(ident.value,
BLOCK.ast.constants,
BLOCK.ast.variables,
BLOCK.ast.procedures);
List<ProcedureAS> procList = new List();
procList.add(proc);
PROCEDUREDECL.ast = procList;
```

  - Beschreibung: Hier wird eine neue Prozedur deklariert. Es muss geprüft werden, ob die Prozedur bereits in der lokalen Umgebung vorhanden ist. Falls ja, führt dies zu einem Fehler. Andernfalls wird eine neue Prozedur erzeugt mit den Elemente des Nichtterminals `BLOCK`. Das Nichtterminal `BLOCK` hat den Typ `BlockContainerAS` (siehe Tabelle 3.7), der dazu dient, Variablen, Konstanten und Prozeduren aufzusammeln. Anschließend wird eine Liste von Prozeduren erzeugt, die die eben erzeugte Prozedur aufnimmt. Sowohl die Umgebung des Nichtterminals `PROCEDUREDECL` als `BLOCK` bekommen die Prozeduren hinzugefügt. Abschließend wird die Liste der Prozeduren dem Nichtterminal `PROCEDUREDECL` hinzugefügt.

- Regel: PROCEDUREDECL1 → 'procedure' ident ';' BLOCK ';' PROCEDUREDECL2
  - ererbtes Attribut:

```
BLOCK.env = PROCEDUREDECL.env
PROCEDUREDECL2.env = PROCEDUREDECL1
```

  - synthetisiertes Attribut:

```
NameableAS element = PROCEDUREDECL.env.lookupLocal(ident.value);
if (element != null) -> Fehler
BLOCK.env.addProcedure(proc);
PROCEDUREDECL2.env.addProcedure(proc);
ProcedureAS proc = new ProcedureAS(ident.value,
BLOCK.ast.constants,
BLOCK.ast.variables,
BLOCK.ast.procedures);
List<ProcedureAS> procList = new List();
procList.addAll(PROCEDUREDECL2.ast);
PROCEDUREDECL.env.addProcedure(proc);
PROCEDUREDECL1.ast = procList;
```

  - Beschreibung: Hier werden mehrere Prozeduren deklariert. Sowohl das Nichtterminal `PROCEDUREDECL2` als auch das Nichtterminal `BLOCK` bekommen die

Umgebung übergeben. Es muss geprüft werden, ob eine Prozedur mit dem Namen `ident.value` in der lokalen Umgebung bereits existiert. Falls ja, führt dies zu einem Fehler. Andernfalls wird eine neue Prozedur erzeugt mit den Informationen des Nichtterminals `BLOCK` (siehe Tabelle 3.7). Es wird eine Liste von Prozeduren erzeugt, der die aktuelle Prozedur und die Prozeduren des Nichtterminals `PROCEDUREDECL2` hinzugefügt werden. Abschließend wird diese Liste dem Nichtterminal `PROCEDUREDECL1` zugewiesen.

- Regel: STATEMENT → 'call' ident
  - ererbtes Attribut: -
  - synthetisiertes Attribut:

    ```
    NameableAS element = STATEMENT.env.lookup(ident.value);
    if (element == null) -> Fehler
    if !(element instanceOf ProcedureAS) -> Fehler
    CallStatementAS stmt = new CallStatementAS(element);
    STATEMENT.ast = stmt;
    ```

  - Beschreibung: Eine `call`-Anweisung kann sich nur auf eine Prozedur beziehen. Aus diesem Grund wird in der Umgebung nach dem Namen `ident.value` nachgesehen. Falls dieser nicht vorhanden ist, wurde kein Element, insbesondere keine Prozedur, mit diesem Namen deklariert. Anschließend muss geprüft werden, falls es ein Element mit diesem Namen gibt, ob es sich dabei um eine Prozedur handelt. Falls nein, handelt es sich um einen Fehler. Wenn beide Prüfungen erfolgreich verliefen, wird ein Aufrufbefehl mit der gefundenen Prozedur erzeugt. Dieser wird dem Nichtterminal `STATEMENT` zugewiesen.

A.4 Erzeugen von neuen Blockniveaus

Während der Attributauswertung wird die Umgebung an genau zwei Stellen neu erzeugt. Diese Regeln werden hier vorgestellt.

- Regel: BLOCK → STATEMENT
  - ererbte Attribute:

    ```
    Envirnoment innerEnv = new Environment();
    innerEnv.setPredecessor(BLOCK.env);
    STATEMENT.env = innerEnv;
    ```

  - synthetisierte Attribute:

    ```
    BlockContainerAS block = new BlockContainerAS(null, null, null,
    STATEMENT.ast);
    BLOCK.ast = block;
    ```

  - Beschreibung: Hier wird eine neue Umgebung erzeugt, da ein neues Blockniveau erreicht wurde. Der neuen Umgebung wird als Vorgänge die alte eingetragen, um so auf die dort deklarierten Elemente zugreifen zu können. Das Nichtterminal `STATEMENT` erhält die neue Umgebung. Ein neues Objekt vom Typ `BlockContainerAS` wird erzeugt und der Befehl hineingepackt.

- Regel: BLOCK → DECL STATEMENT
  - ererbte Attribute:

```
Environment innerEnv = new Environment();
innerEnv.setPredecessor(BLOCK.env);
DECL.env = innerEnv;
STATEMENT.env = innerEnv;
```

– synthetisierte Attribute:

```
BlockContainerAS block = new BlockContainerAS(
DECL.ast.constants,
DECL.ast.variabeles,
DECL.ast.procedures,
STATEMENT.ast);
BLOCK.ast = block;
```

– Beschreibung: Eine neue Umgebung wird erzeugt. Die alte Umgebung wird zum Vorgänger. Sowohl das Nichtterminal `DECL` als auch `STATEMENT` bekommen die neue Umgebung übergeben. Es wird ein neues Objekt vom Typ `BlockContainerAS` erzeugt, das die Inhalte der Deklaration und des Befehls übergeben bekommt.

A.5 Semantik der restlichen Grammatikregeln

Die Semantik der restlichen Grammatikregeln beinhalten lediglich das Zusammenbauen der Elemente aus der abstrakten Syntax. Aus diesem Grund soll ein Beispiel genügen.

- Regel: STATEMENT1 → 'if' CONDITION 'then' STATEMENT2
  - ererbte Attribute:

    ```
    CONDITION.env = STATEMENT1.env;
    STATEMENT2.env = STATEMENT1.env;
    ```

  - synthetisierte Attribute:

    ```
    IfStatementAS stmt = new IfStatementAS(CONDITION.ast,
    STATEMENT2.ast);
    STATEMENT1.ast = condition;
    ```

  - Beschreibung: Die Umgebung wird an alle Nichtterminale weitergegeben. Es wird ein Bedingungsbefehl aus der Bedingung und aus dem Element von `STATEMENT2.ast` zusammengebaut und an das Nichtterminal `STATEMENT1` übergeben.

B Abbildung der originalen OCL-Grammatik auf die angepassten Grammatik

Im Folgenden wird die Abbildung der Regeln der originalen OCL-Grammatik aus [OCL2] auf die angepasste Grammatik des OCL-Parsers beschrieben. Im ersten Abschnitt werden die Regeln der originalen Grammatik auf die Regeln des OCL-Parsers abgebildet. Im zweiten Abschnitt werden neu hinzugefügte Regeln behandelt, die nicht Bestandteil der originalen Grammatik sind.

B.1 Abbildung der Regeln der originalen Grammatik

Jede Beschreibung beginnt mit der Darstellung einer Regel aus der originalen Grammatik. Darunter werden die Regeln, sofern es welche gibt, aufgezeigt, die die Entsprechung in der angepassten Grammatik darstellen. Zum Schluss werden einige Anmerkungen zu der jeweiligen Abbildung gemacht.

Zu beachten ist, dass die Regeln aus [OCL2] unverändert übernommen wurden. Lediglich die Kennzeichnung der Morpheme wurde etwas geändert. Statt sie mit den Zeichen `’ ’` einzuschließen, werden sie nun mit den Zeichen `" "` gekennzeichnet.

Die Regeln der angepassten Grammatik wurden aus der SableCC-Spezifikationsdatei übernommen. Aus diesem Grund werden die Alternativnamen und Schlüsselwörter mitgeführt. Einige lange Morphemnamen wurden durch ihre jeweilige Zeichenkette ersetzt. So wurde beispielweise das Morphem `open_paren` durch das Symbol `(` ersetzt. Diese Morpheme werden durch die Zeichen `" "` eingeschlossen.

OCL-Spez.	`[A] packageDeclarationCS ::= "package" pathNameCS contextDeclCS* "endpackage"` `[B] packageDeclarationCS ::= contextDeclCS*`
SableCC-Spez.	`packaged_constraint_list_cs = "package" path_name_cs context_declaration_cs* "endpackage"`
Bemerkung	Regel `[A]` wird aus der originalen OCL-Grammatik übernommen. Regel `[B]` dagegen nicht. So ist es nicht möglich einen OCL-Ausdruck ohne eine Paketdeklaration zu schreiben.

OCL-Spez.	`[A] contextDeclarationCS ::= attrOrAssocContextCS`
	`[C] contextDeclarationCS ::= classifierContextDeclCS`
	`[D] contextDeclarationCS ::= operationContextDeclCS`

SableCC-Spez.	keine direkte Entsprechung
Bemerkung	Die originalen Grammatikregeln an sich existieren so nicht in der überarbeiteten Grammatik. Die Regeln in der originalen OCL-Grammatik dienen lediglich als Weiterleitung. Diese Weiterleitung wurde eingespart. Statt dessen wurden die Nichtterminale `attrOrAssocContextCS`, `classifierContextDeclCS` und `operationContextDeclCS` direkt durch ihre Definitionen ersetzt.

OCL-Spez.	`attrOrAssocContextCS ::= "context" pathNameCS "::" simpleName ":" typeCS initOrDerValueCS`

SableCC-Spez.	`context_declaration_cs = {attr_or_assoc} "context" path_name_cs context_type?`
	`init_or_der_value_cs+`
	
	`context_type = ":" type_specifier#chain`

Bemerkung	Die Symbole `pathNameCS "::" simpleName"` wurden zu `path_name_cs` zusammengefasst, ebenso wurden die Symbole `":" typeCS` zu `context_type?`. Zu beachten ist, dass in der originalen Grammatik immer eine Typangabe folgen muss, in der angepassten Grammatik dagegen nicht.

OCL-Spez.	`[A] initOrDerValueCS[1] ::= "init" ":" OclExpression initOrDerValueCS[2]?` `[B] initOrDerValueCS[1] ::= "derive" ":" OclExpression initOrDerValue[2]?`
SableCC-Spez.	`init_or_der_value_cs = {init} "init" ":" ocl_expression_cs` `init_or_der_value_cs = {derive} "derive" ":" ocl_expression_cs`
Bemerkung	Im Prinzip wurden beide originale Regeln übernommen. Die Rekursion wurde eingespart und auf die Regel `context_declaration_cs` verschoben (Annotation `+`).

OCL-Spez.	`classifierContextDeclCS ::= "context" pathNameCS invOrDefCS`
SableCC-Spez.	`context_declaration_cs = {classifier} "context" path_name_cs classifier_constraint_cs+`
Bemerkung	Die originale Regel wurde im Prinzip übernommen. Das Nichtterminal `classifier_constraint_cs` in der angepassten Grammatik fasst ebenso wie das Nichtterminal `invOrDefCS` verschiedene Bedingungen zusammen.

OCL-Spez.	`[A] invOrDefCS[1] ::= "inv" (simpleNameCS)? ":" OclExpressionCS invOrDefCS[2]` `[B] invOrDefCS[1] ::= "def" (simpleNameCS)? ":" defExpressionCS invOrDefCS[2]`

SableCC-Spez.	`classifier_constraint_cs = {inv} "inv" simple_name? ":" ocl_expression_cs`
	`classifier_constraint_cs = {def_attribute} "def" simple_name? ":" simple_name ":" type_specifier "=" ocl_expression_cs`
	
	`classifier_constraint_cs = {def_operation} "def" simple_name? ":" simple_name operation_signature_cs "=" ocl_expression_cs`

Bemerkung	Regel `[A]` der originalen Grammatik wurde genauso übernommen. Regel `[B]` findet sich in den beiden letzten Regeln der angepassten Grammatik wieder. Das Nichtterminal `defExpressionCS` wurde aufgelöst und statt dessen direkt in die Regeln integriert. Dadurch wird eine Container-Klasse vermieden (siehe Abschnitt 3.4.1 auf Seite 51). Die Rekursion in den originalen Regeln wurde auf das Nichtterminal `context_declaration_cs` in der angepassten Grammatik verschoben (Annotation `+`).

OCL-Spez.	`[A] defExpressionCS ::= VariableDeclarationCS "=" OclExpression` `[B] defExpressionCS ::= operationCS "=" OclExpression`

SableCC-Spez.	`classifier_constraint_cs = {def_attribute} "def" simple_name? ":" simple_name ":" type_specifier "=" ocl_expression_cs`
	
	`classifier_constraint_cs = {def_operation} "def" simple_name? ":" simple_name operation_signature_cs "=" ocl_expression_cs`

Bemerkung	Wie für das Grammatiksymbol `invOrDefCS` beschrieben, wurde das Nichtterminal `defExpressionCS` aufgelöst. Hier sind der Vollständigkeit die beiden Regeln der angepassten Grammatik noch einmal aufgeführt. Das Nichtterminal `VariableDeclarationCS` der originalen Grammatik wurde zum Nichtterminal `type_specifier` der angepassten Grammatik. Die originale Grammatik lässt an dieser Stelle eine Zuweisung eines OCL-Ausdruckes zu, was zu falschen Ausdrücken führt.

OCL-Spez.	`operationContextDeclCS := "context" operationCS prePostOrBodyDeclCS`
SableCC-Spez.	`context_declaration_cs = {operation} "context" path_name_cs operation_signature_cs operation_constraint_cs+`
Bemerkung	Das Nichtterminal `operationCS` wurde teilweise durch seine Definition direkt ersetzt. `operationCS` führt den Namen der Operation, die Parameter und den Rückgabewert ein. Der Name der Operation wird in der angepassten Grammatik mit dem Nichtterminal `path_name_cs` beschrieben. Die Parameter und der Rückgabewert der Operation wird durch `operation_signature_cs` ausgedrückt. Das Nichtterminal `prePostOrBodyDeclCS` wird durch `operation_constraint_cs` ersetzt.

OCL-Spez.	`[A] prePostOrBodyDeclCS[1] ::= "pre" (simpleNameCS)? ":" OclExpressionCS prePostOrBodyDeclCS[2]?`
	`[B] prePostOrBodyDeclCS[1] ::= "post" (simpleNameCS)? ":" OclExpressionCS prePostOrBodyDeclCS[2]?`
	`[C] prePostOrBodyDeclCS[1] ::= "body" (simpleNameCS)? ":" OclExpressionCS prePostOrBodyDeclCS[2]?`

SableCC-Spez.	`operation_constraint_cs = {full} op_constraint_stereotype_cs simple_name? ":" ocl_expression_cs`
	`operation_constraint_cs = {empty} op_constraint_stereotype_cs ":"`
	`op_constraint_stereotype_cs = {pre} "pre" #chain| {post} "post"#chain
		{body} "body"#chain`

Bemerkung	Die drei Schlüsselwörter `pre`, `post` und `body` wurden in einem Nichtterminal `op_constraint_stereotype_cs` zusammengefasst. Die weiteren Symbole der originalen Regel wurden übernommen, bis auf das zweite Vorkommen des Nichtterminals `prePostOrBodyDeclCS`. Diese Rekursion ergibt sich über die Regel `context_declaration_cs` (Alternative `operation`) der angepassten Grammatik. Die Alternative `empty` der Regel `operation_constraint_cs` der angepassten Grammatik kommt in der originalen Grammatik so nicht vor. Diese Regel dient lediglich dazu, eines der zuvor erwähnten Schlüsselwörter ohne OCL-Ausdruck schreiben zu können (siehe Abschnitt 6.2.1 auf Seite 103).

OCL-Spez.	`[A] operationCS ::= pathNameCS "::" simpleNameCS "(" parametersCS? ")" ":" typeCS?` `[B] operationCS ::= simpleNameCS "(" parametersCS? ")" ":" typeCS?`
SableCC-Spez.	`operation_signature_cs = "(" formal_parameter_list_cs? ")" operation_return_type_specifier_cs?` `operation_return_type_specifier_cs = ":" type_specifier#chain`
Bemerkung	Die Symbole `pathNameCS "::" simpleNameCS` der originalen Grammatik werden zum Nichtterminal `path_name_cs` der angepassten Grammatik (siehe Nichtterminal `context_declaration_cs`, Alternative `operation` der angepassten Grammatik). Die Symbole `"(" parametersCS? ")"` werden zu den Symbolen `open_paren formal_parameter_list_cs? close_paren` der angepassten Grammatik. Hier lässt die originale Grammatik Zuweisungen in den Parametern zu, was aber nicht gewollt ist. Das Nichtterminal `formal_parameter_cs` der angepassten Grammatik erzeugt eine Parameterliste, deren Parameter genau einen Typen haben. Die Symbole `":" typeCS?` werden zum Symbol `operation_return_type_specifier_cs?` in der angepassten Grammatik.

OCL-Spez.	`parametersCS[1] ::= VariableDeclarationCS ("," parametersCS[2])?`

SableCC-Spez.	`formal_parameter_list_cs = formal_parameter_cs formal_parameter_enum_cs*`
	`formal_parameter_enum_cs = "," formal_parameter_cs#chain`
	`formal_parameter_cs = simple_name formal_parameter_type_specifier`
	`formal_parameter_type_specifier = ":" type_specifier#chain`

Bemerkung	Das Nichtterminal `parametersCS` der originalen Grammatik entspricht in etwa den vier angegebenen Regeln der angepassten Grammatik. Allerdings kann das Nichtterminal `VariableDeclarationCS` auch Parameter erzeugen, die keinen Typen haben und denen ein OCL-Ausdruck zugewiesen werden kann. Das ist mit den angegebenen Regeln der angepassten Grammatik nicht möglich. Hier muss jeder Parameter genau einen Typen besitzen und er darf keinen OCL-Ausdruck zugewiesen bekommen.

OCL-Spez.	`ExpressionInOclCS ::= OclExpressionCS`
SableCC-Spez.	keine Entsprechung
Bemerkung	Im Abschnitt *Inherited attributes* wird verlangt, dass in der Umgebung die Variable `self` und gegebenenfalls die Variable `result` erzeugt wird. Dieses findet sich im Aspekt `ContextASMComputation` (siehe auch Abschnitt 6.3.7 auf Seite 118).

OCL-Spez.	`[A] OclExpressionCS ::= PropertyCallExpCS`
	`[B] OclExpressionCS ::= VariableExpCS`
	`[C] OclExpressionCS ::= LiteralExpCS`
	`[D] OclExpressionCS ::= LetExpCS`
	`[E] OclExpressionCS ::= OclMessageExpCS`
	`[F] OclExpressionCS ::= IfExpCS`

SableCC-Spez.	`ocl_expression_cs = {let} let_exp_cs#chain` `ocl_expression_cs = {logical} logical_exp_cs#chain`
Bemerkung	Eine direkte Entsprechung gibt es für die Regel `[D]` der originalen Grammatik (siehe erste angegebene Regel der angepassten Grammatik). Alle anderen Regeln der originalen Grammatik finden sich in der zweiten Regel der angepassten Grammatik wieder.

OCL-Spez.	`VariableExpCS ::= simpleNameCS`
SableCC-Spez.	keine Entsprechung
Bemerkung	Das Nichtterminal `VariableExpCS` besitzt in der angepassten Grammatik keine Entsprechung. Mit dem Nichtterminal der originalen Grammatik sollen Variablen in Ausdrücken erfasst werden. Diese werden durch das Nichtterminal `property_call_exp_cs` der angepassten Grammatik abgedeckt. Allerdings kann ein Ausdruck, der durch das Nichtterminal `property_call_exp_cs` generiert wird, auch für den normalen Attributaufruf stehen. Aus diesem Grund muss dies semantisch ausgewertet werden.

OCL-Spez.	`simpleNameCS ::= <String>`
SableCC-Spez.	`path_name_cs = identifier_cs identifier_enum_cs*`
Bemerkung	Ein einfacher Name wird immer mit dem Nichtterminal `path_name_cs` der angepassten Grammatik abgedeckt, auch wenn dieser mehrere Elemente - durch den Doppelpunkt getrennt- zulässt. Während der semantischen Analyse muss dies ausgewertet werden. Das Nichtterminal `identifier_cs` bildet auf einen Identifizierer ab. In der Grammatik werden diese mit ein paar Schlüsselwörtern, die ebenfalls als Identifizierer in Frage kommen könnten, zusammengefasst.

OCL-Spez.	`pathNameCS ::= simpleName ("::" pathNameCS )?`

SableCC-Spez.	`path_name_cs = identifier_cs identifier_enum_cs*`
	`identifier_enum_cs = "::" identifier_cs#chain`
	`identifier_cs = {simple} simple_name#chain`
	`identifier_cs = {ocl_op_name} ocl_op_name#chain`
	`identifier_cs = collection_type_identifier_cs#chain`
	`ocl_op_name = {kind_of} "oclIsKindOf"#chain`
	`ocl_op_name = {type_of} "oclIsTypeOf"#chain`
	`ocl_op_name = {as_type} "oclAsType"#chain`
	`collection_type_identifier_cs = {set} "Set"#chain`
	`collection_type_identifier_cs = {bag} "Bag"#chain`
	`collection_type_identifier_cs = {sequence} "Sequence"#chain`
	`collection_type_identifier_cs = {collection} "Collection"#chain`
	`collection_type_identifier_cs = {orderedset} "OrderedSet"#chain`

Bemerkung	Der Pfadname wurde in der angepassten Grammatik so übernommen. Es fehlt lediglich die Rekursion, die über den Sternoperator realisiert wird (siehe Nichtterminal `path_name_cs` der angepassten Grammatik). Das Nichtterminal `simpleName` der originalen Grammatik wurde zum Nichtterminal `identifier_cs` der angepassten Grammatik. Dieses Nichtterminal vereint mehrere andere Klassen von Morphemen wie die speziellen OCL-Operationsnamen und die Kollektionstypen. Dies resultiert daraus, dass die Kollektionstypen in den Kollektionsliteralen explizit verwendet werden, aber ein Identifizierer kann auch genau so lauten wie ein Kollektionstyp, beispielsweise könnte eine Klasse (oder ein *Type* im Pivotmodell) ein Attribut (oder ein *Property* im Pivotmodell) mit dem Namen `bag` besitzen. Dieses Attribut wäre dann als Identifizierer zu werten und nicht als Schlüsselwort. Die speziellen OCL-Operationen wurden zwar als Schlüsselwörter eingeführt, aber sie könnten auch weggelassen und zu einem normalen `simple_name`-Morphem werden.

OCL-Spez.	`[A] LiteralExpCS ::= EnumLiteralExpCS`
	`[B] LiteralExpCS ::= CollectionLiteralExpCS`
	`[C] LiteralExpCS ::= TupleLiteralExpCS`
	`[D] LiteralExpCS ::= PrimitiveLiteralExpCS`

SableCC-Spez.	`literal_exp_cs = {collection_literal} collection_literal_exp_cs#chain`
	`literal_exp_cs = {tuple} tuple_literal_exp_cs#chain`
	`literal_exp_cs = {primitive} primitive_literal_exp_cs#chain`

Bemerkung	Bis auf eine Ausnahme wurden die Regeln der originalen Grammatik übernommen. Es gibt kein Nichtterminal `EnumLiteralExpCS`. Das liegt daran, dass ein Aufzählungsliteral von einem Attributnamen nicht zu unterscheiden ist (beide können Elemente enthalten, die mit dem doppelten Doppelpunkt `::` getrennt werden). Das würde zu einem Konflikt führen. Daher werden die Aufzählungsliterale erst zur semantischen Prüfung erkannt (siehe Abschnitt 6.3.19 auf Seite 124).

OCL-Spez.	`EnumLiteralExpCS ::= pathNameCS "::" simpleNameCS`
SableCC-Spez.	keine Entsprechung
Bemerkung	Das Nichtterminal `EnumLiteralExpCS` gibt es in der angepassten Grammatik so nicht. Die Ausdrücke der Aufzählungsliterale werden über das Nichtterminal `property_call_exp_cs` abgedeckt.

OCL-Spez.	`CollectionLiteralExpCS ::= CollectionTypeIdentifierCS "{" CollectionLiteralPartsCS? "}"`
SableCC-Spez.	`collection_literal_exp_cs = collection_type_identifier_cs "{" collection_literal_parts_cs? "}"`
Bemerkung	Die Regel der originalen Grammatik wurde genauso übernommen, lediglich die Namen der Nichtterminale haben sich leicht geändert.

OCL-Spez.	`[A] CollectionTypeIdentifierCS ::= "Set"`
	`[B] CollectionTypeIdentifierCS ::= "Bag"`
	`[C] CollectionTypeIdentifierCS ::= "Sequence"`
	`[D] CollectionTypeIdentifierCS ::= "Collection"`
	`[E] CollectionTypeIdentifierCS ::= "OrderedSet"`

SableCC-Spez.	`collection_type_identifier_cs = {set} "Set"#chain`
	`collection_type_identifier_cs = {bag} "Bag"#chain`
	`collection_type_identifier_cs = {sequence} "Sequence"#chain`
	`collection_type_identifier_cs = {collection} "Collection"#chain`
	`collection_type_identifier_cs = {orderedset} "OrderedSet"#chain`

Bemerkung	Die Regeln der originalen Grammatik wurden genauso übernommen.

OCL-Spez.	`CollectionLiteralPartsCS[1] ::= CollectionLiteralPartCS ( "," CollectionLiteralPartsCS[2] )?`
SableCC-Spez.	`collection_literal_parts_cs = collection_literal_part_cs collection_literal_part_enum_cs*` `collection_literal_part_enum_cs = "," collection_literal_part_cs#chain`
Bemerkung	Die Regel der originalen Grammatik wurde in zwei Regeln aufgespaltet. Dadurch entfällt die Rekursion.

OCL-Spez.	`[A] CollectionLiteralPartCS ::= CollectionRangeCS` `[B] CollectionLiteralPartCS ::= OclExpressionCS`
SableCC-Spez.	`collection_literal_part_cs = {range} collection_range_cs #chain` `collection_literal_part_cs = {single_exp} ocl_expression_cs`
Bemerkung	Beide Regeln der originalen Grammatik wurden genauso übernommen.

OCL-Spez.	`CollectionRange ::= OclExpressionCS[1] "," OclExpressionCS[2]`
SableCC-Spez.	`collection_range_cs = ocl_expression_cs ".." ocl_expression_cs;`
Bemerkung	Im Prinzip entspricht die Regel der originalen Grammatik der Regel der angepassten Grammatik, allerdings wird ein Kollektionsbereich nicht mit einem Komma `,`, sondern mit zwei Punkten `..` angegeben.

OCL-Spez.	`[A] PrimitiveLiteralExpCS ::= IntegerLiteralExpCS`
	`[B] PrimitiveLiteralExpCS ::= RealLiteralExpCS`
	`[C] PrimitiveLiteralExpCS ::= StringLiteralExpCS`
	`[D] PrimitiveLiteralExpCS ::= BooleanLiteralExpCS`

SableCC-Spez.	`primitive_literal_exp_cs = {numeric} numeric_literal_exp_cs#chain`
	`primitive_literal_exp_cs = {string} string_literal_exp_cs#chain`
	`primitive_literal_exp_cs = {boolean} boolean_literal_exp_cs#chain`

Bemerkung	Im Prinzip wurden die Regeln der originalen Grammatik übernommen. Allerdings wurde in der angepassten Grammatik das Nichtterminal `numeric_literal_exp_cs` eingeführt, das die Nichtterminale `RealLiteralExpCS` und `IntegerLiteralExpCS` der originalen Grammatik zusammenfasst.

OCL-Spez.	`TupleLiteralExpCS ::= "Tuple" "{" variableDeclarationListCS "}"`

SableCC-Spez.	`tuple_literal_exp_cs = {tuple} tuple "{" initialized_variable_list_cs "}"`
	`initialized_variable_list_cs = initialized_variable_cs initialized_variable_enum_cs*`
	`initialized_variable_enum_cs = "," initialized_variable_cs#chain`
	`initialized_variable_cs = initialized_parameter_cs "=" ocl_expression_cs#nocreate`
	`initialized_parameter_cs = {formal} formal_parameter_cs#nocreate | {simple} simple_name`

Bemerkung	Im Prinzip wurde die Regel der originalen Grammatik übernommen. Die Variablenliste der originalen Grammatik stimmt nicht ganz, da durch das Nichtterminal `variableDeclarationListCS` auch Variablendefinitionen eingebracht werden können (diese Liste wird auf das Nichtterminal `VariableDeclarationCS` der originalen Grammatik abgebildet), die keinen initialen OCL-Ausdruck besitzen. Statt dessen wird diese Prüfung auf die semantische Analyse verschoben. In der angepassten Grammatik wird dies auf den Parser verschoben. Aus diesem Grund wurden die initialisierten Variablen (siehe Nichtterminal `initialized_variable_cs`) eingeführt, die immer einen OCL-Ausdruck zugewiesen bekommen. Da ein initialisierter Parameter auch ein formaler Parameter (siehe Nichtterminal `formal_parameter_cs`) sein kann, kann auch ein Typ für die Variable angegeben werden. Die restlichen Regeln der angepassten Grammatik dienen lediglich dazu, eine Liste von Parameter zu bauen, die mindestens ein Element enthält.

OCL-Spez.	`IntegerLiteralExpCS ::= <String>`
SableCC-Spez.	`numeric_literal_exp_cs = {integer} integer_literal`
Bemerkung	Wie zu dem Nichtterminal `PrimitiveLiteralExpCS` der originalen Grammatik geschrieben, gibt es in der angepassten Grammatik kein Nichtterminal `IntegerLiteralExpCS`. Statt dessen werden die ganzen und reellen Zahlen durch das Nichtterminal `numeric_literal_exp_cs` gefasst. Hier wird die entsprechende Regel der angepassten Grammatik gezeigt. Sie zeigt nicht einfach auf eine beliebige Zeichenkette, sondern auf ein Zahlenliteral (siehe das Morphem `integer_literal`).

OCL-Spez.	`RealLiteralExpCS ::= <String>`
SableCC-Spez.	`numeric_literal_exp_cs = {real}<RealLiteralExpAS> real_literal`
Bemerkung	Wie zu dem Nichtterminal `PrimitiveLiteralExpCS` der originalen Grammatik geschrieben, gibt es in der angepassten Grammatik kein Nichtterminal `RealLiteralExpCS`. Statt dessen werden die reellen Zahlen durch das Nichtterminal `numeric_literal_exp_cs` gefasst. Hier wird die entsprechende Regel der angepassten Grammatik gezeigt. Sie zeigt nicht einfach auf eine beliebige Zeichenkette, sondern auf ein Literal, dass eine reelle Zahl repräsentiert. (siehe das Morphem `real_literal`).

OCL-Spez.	`StringLiteralExpCS ::= "<String>"`
SableCC-Spez.	`string_literal_exp_cs = string_literal`
Bemerkung	Diese Regel wurde übernommen.

OCL-Spez.	`[A] BooleanLiteralExpCS ::= "true"` `[B] BooleanLiteralExpCS ::= "false"`
SableCC-Spez.	`boolean_literal_exp_cs = {false} "false" | {true} "true"`
Bemerkung	Diese Regeln wurden übernommen.

OCL-Spez.	`[A] PropertyCallExpCS ::= ModelPropertyCallExpCS` `[B] PropertyCallExpCS := LoopExpCS`
SableCC-Spez.	keine Entsprechung
Bemerkung	Das Nichtterminal `PropertyCallExpCS` fasst lediglich zwei andere Nichtterminale zusammen. In der angepassten Grammatik werden diese Regeln vermieden.

OCL-Spez.	`[A] LoopExpCS ::= IteratorExpCS` `[B] LoopExpCS ::= IterateExpCS`
SableCC-Spez.	keine Entsprechung
Bemerkung	Das Nichtterminal `LoopExpCS` fasst lediglich zwei andere Nichtterminale zusammen. In der angepassten Grammatik werden diese Regeln vermieden.

OCL-Spez.	`[A] IteratorExpCS ::= OclExpressionCS[1] "->" simpleNameCS "(" (VariableDeclarationCS[1], ( "," VariableDeclarationCS[2])? "|" )? OclExpressionCS[2] ")"`

SableCC-Spez.	`postfix_exp_cs = {arrowright_iterator} postfix_exp_cs "->" iterator_operation "(" iterator_var_cs? ocl_expression_cs ")"`
	
	`iterator_var_cs = parameter_list_cs "|"#chain`
	`parameter_list_cs = parameter_cs parameter_enum*`
	`parameter_enum = "," parameter_cs#chain`
	`parameter_cs = {formal_parameter}<VariableAS> formal_parameter_cs`
	`parameter_cs ={simple_name}<VariableAS> simple_name`

Bemerkung	Diese Regel (originale Grammatik) steht für die Iteratoren-Ausdrücke. In der Regel kann jede Zeichenketten (vgl. Nichtterminal `simpleNameCS`) als Iterator-Operation verwendet werden. Tatsächlich kommen aber nur die Zeichenketten `collect`, `select` etc. in Frage. Das ist der Grund, warum in der angepassten Grammatik das Morphem `iterator_operation` definiert wird. Dieses steht genau für die zulässigen Operationsnamen. In der originalen Grammatik sind lediglich zwei Iteratoren-Variablen zugelassen. In der angepassten Grammatik besteht dagegen keine Beschränkung. Die Variablen dürfen keinen initialen OCL-Ausdruck besitzen, was in der originalen Regel nicht beachtet wird. In der angepassten Grammatik werden wieder die formalen Parameter (vgl. Nichtterminal `formal_paramter_cs` in der Regeldefinition des Nichtterminals `parameter_cs`) verwendet, um keine initialen OCL-Ausdrücke zuzulassen. Das Vorkommen des Nichtterminals `OclExpressionCS[2]` der originalen Grammatik findet sich auch so in der angepassten Grammatik wieder (vgl. Nichtterminal `postfix_exp_cs`). Zum Nichtterminal `OclExpressionCS[1]` der originalen Regel muss etwas mehr ausgeholt werden. Für die Iterator-Ausdrücke bedeutet das Nichtterminal `OclExpression[1]`, dass jeder beliebige OCL-Ausdruck als Quellausdruck verwendet werden darf. In der angepassten Grammatik darf dagegen nicht jeder OCL-Ausdruck verwendet werden. Ausgeschlossen sind die mathematischen und die `let`-Ausdrücke. Der Grund liegt in der Mehrdeutigkeit der originalen Grammatik. Die originale Grammatik erlaubt es, zwei `let`-Ausdrücke, die Iterator-Ausdrücke beinhalten, auf zwei unterschiedliche Arten zu erzeugen. Im Prinzip geht es darum, welcher Ausdruck in einem Iterator-Ausdruck zu einem Quellausdruck werden soll. Dazu ein Beispiel: `let a='5' in a -> collect(b)`. Bezieht sich der Pfeiloperator auf das Zeichen `a` oder auf den gesamten `let`-Ausdruck? In der originalen Grammatik ist dies durch die Mehrdeutigkeit nicht gelöst. In der Arbeit [ANS] wurde angenommen, dass sich der Pfeiloperator, bezogen auf das Beispiel, auf den Ausdruck `a` bezieht. Um den Pfeiloperator auf den `let`-Ausdruck anzuwenden, muss dieser in Klammern gesetzt werden. Genau dieses Verhalten wird in der angepassten Grammatik umgesetzt. Eine weitere Besonderheit betrifft die mathematischen Ausdrücke. Diese werden in der originalen Grammatik ungewöhnlich formuliert (siehe Nichtterminal `OperationCallExpCS` Alternative `[A]`). Dadurch ergibt sich auch hier wieder eine Mehrdeutigkeit, die durch die Verwendung der Klammern aufgelöst wird.

OCL-Spez.	`[B] IteratorExpCS ::= OclExpressionCS "." simpleNameCS "(" argumentsCS ")"`
SableCC-Spez.	`postfix_exp_cs = {dot_operation_call} postfix_exp_cs "." identifier_cs atpre? "(" actual_parameter_list_cs? ")"`
Bemerkung	Die Regel der originalen Grammatik steht für einen Operationsaufruf auf einer Menge von Elementen. In diesem Fall wird die `collect`-Operation eingeschoben (siehe Abschnitt 6.3.11 auf Seite 120). Der Parser erkennt dies mit dem normalen Operationsaufruf. Während der semantischen Auswertung muss geprüft werden, ob der Quellausdruck eine Menge darstellt oder nicht.

OCL-Spez.	`[C] IteratorExpCS ::= OclExpressionCS "." simpleNameCS`
SableCC-Spez.	`postfix_exp_cs = {dot_property_call} postfix_exp_cs "." identifier_cs atpre?`
Bemerkung	Das Anliegen der Regel der originalen Grammatik besteht darin, ein Attribut auf einer Menge von Elemente aufzurufen. Dabei wird die `collect`-Operation eingeschoben (siehe Abschnitt 6.3.11 auf Seite 120). In der angepassten Grammatik wird dies durch einen "normale" Attributzugriff abgebildet. Während der semantischen Analyse muss geprüft werden, ob der Quellausdruck eine Menge von Elementen darstellt.

OCL-Spez.	`[D] IteratorExpCS ::= OclExpressionCS "." simpleNameCS ("[" argumentsCS "]" )?`
SableCC-Spez.	`postfix_exp_cs = {dot_property_assoc_call} postfix_exp_cs "." identifier_cs "[" actual_parameter_list_cs "]" atpre?`
Bemerkung	Die Regel der originalen Grammatik dient dazu, Ausdrücke zu fassen, die einen Attributzugriff mit Qualifizierungen auf Mengen darstellen. In der angepassten Grammatik wird dies auf einen "normalen" Attributzugriff mit Qualifizierung abgebildet. Während der semantischen Analyse muss geprüft werden, ob der Quellausdruck eine Menge von Elemente darstellt.

OCL-Spez.	`[E] IteratorExpCS ::= OclExpressionCS "." simpleNameCS ("[" argumentsCS "]" )?`
SableCC-Spez.	keine Entsprechung
Bemerkung	Die Regel steht für die Anwendung auf Assoziationsklassen. Assoziationsklassen werden im PivotModell nicht unterstützt. Daher wird diese Regel ignoriert.

OCL-Spez.	`IterateExpCS ::= OclExpressionCS[1] "->" "iterate"` `"(" (VariableDeclarationCS[1] ";" )? VariableDeclarationCS[2] "|" OclExpressionCS[2] ")"`

SableCC-Spez.	`postfix_exp_cs = {arrowright_iterate} postfix_exp_cs "->" "iterate"`
	`"(" iterate_var_cs? initialized_variable_cs "|" ocl_expression_cs ")"`
	
	`iterate_var_cs = formal_parameter_cs ";"#nocreate`

Bemerkung	Die Regel der originalen Grammatik drückt einen *Iterate*-Ausdruck aus. Er kann eine Iterator-Variable (vgl. Nichtterminal `VariableDeclarationCS[1]`) und muss eine Rückgabevariable (vgl. Nichtterminal `VariableDeclarationCS[2]`) besitzen. Beide Variablen finden sich in der angepassten Regel wieder, wobei gefordert ist, dass die Rückgabevariable einen initialen OCL-Ausdruck hat (das ist nicht strikt in der originalen Regel gefordert, da das Nichtterminal `VariableDeclarationCS` auch Variablenausdrücke erzeugen kann, die keinen initialen OCL-Ausdruck haben). Das Nichtterminal `OclExpressionCS[1]` verursacht die gleichen Probleme wie bei der Definition des Nichtterminals `IteratorExpCS` (Stichwort: Mehrdeutigkeit).

OCL-Spez.	`VariableDeclarationCS ::= simpleNameCS ( ":" typeCS )? ( "=" OclExpressionCS )?`

SableCC-Spez.	`formal_parameter_cs = simple_name formal_parameter_type_specifier`
	`formal_parameter_type_specifier = ":" type_specifier#chain`
	`initialized_parameter_cs = {formal} formal_parameter_cs#nocreate`
	`initialized_parameter_cs = {simple} simple_name`

Bemerkung	Das Nichtterminal `VariableDeclarationCS` der originalen Grammatik fasst mehrere Kombinationen an Variablen bzw. an Parametern zusammen. In der angepassten Grammatik gibt es dagegen drei Regeln, die alle drei möglichen Fälle abdecken. An vielen Stellen wird nur eine der Varianten benötigt.

OCL-Spez.	`[A] typeCS ::= pathNameCS`
	`[B] typeCS ::= collectionTypeCS`
	`[C] typeCS ::= tupleTypeCS`

SableCC-Spez.	`type_specifier = {simple} path_name_cs#chain`
	`type_specifier = {collection_type} collection_type_specifier_cs#chain`
	`type_specifier = {tuple_type} tuple_type_specifier_cs#chain`

Bemerkung	Die Regel der originalen Grammatik wurde mit Umbenennung der Nichtterminale übernommen.

OCL-Spez.	`collectionTypeCS ::= collectionTypeIdentifierCS "(" typeCS ")"`
SableCC-Spez.	`collection_type_specifier_cs = collection_type_identifier_cs "(" type_specifier ")"`
Bemerkung	Die Regel der originalen Grammatik wurde mit Umbenennung der Nichtterminale übernommen.

OCL-Spez.	`tupleTypeCS ::= "Tuple" "(" variableDeclarationListCS? ")"`
SableCC-Spez.	`tuple_type_specifier_cs = "TupleType" "(" formal_parameter_list_cs? ")"`
Bemerkung	Im Grunde wurde die Regel der originalen Grammatik mit Umbenennung der Nichtterminale übernommen. Eine Ausnahme bildet das Nichtterminal `variableDeclarationListCS`. Dieses lässt Ausdrücke mit initialem OCL-Ausdruck zu, was aber nicht zugelassen ist. Das Nichtterminal `formal_parameter_list_cs` lässt dagegen ausschließlich Variablen mit Typangabe zu.

OCL-Spez.	`variableDeclarationListCS[1] ::= VariableDeclarationCS ( "," variableDeclarationListCS[2] )?`
SableCC-Spez.	`formal_parameter_list_cs = formal_parameter_cs formal_parameter_enum_cs*` `initialized_variable_list_cs = initialized_variable_cs initialized_variable_enum_cs*`
Bemerkung	Das Nichtterminal `variableDeclarationListCS` gibt es so nicht in der angepassten Grammatik. Dieses Nichtterminal erzeugt eine Liste von Variablen, die einen optionalen Typen und OCL-Ausdruck haben können. An vielen Stellen ist aber ein initialer OCL-Ausdruck oder eine Typangabe explizit vorgeschrieben, so dass dies in zwei Varianten aufgespaltet wurde.

OCL-Spez.	`[A] ModelPropertyCallExpCS ::= OperationCallExpCS`
	`[B] ModelPropertyCallExpCS ::= AttributeCallExpCS`
	`[C] ModelPropertyCallExpCS ::= NavigationCallExpCS`

SableCC-Spez.	keine Entsprechung
Bemerkung	Diese Weiterleitungsregeln der originalen Grammatik haben keine Entsprechung in der angepassten Grammatik. Die Ausdrücke in der angepassten Grammatik werden in den Nichtterminalen `postfix_exp_cs` und `property_call_exp_cs` zusammengefasst.

OCL-Spez.	`[A] OperationCallExpCS ::= OclExpressionCS[1] simpleNameCS OclExpressionCS[2]` `[H] simpleNameCS OclExpressionCS`

SableCC-Spez.	`logical_exp_cs = {logical} logical_exp_cs "implies" simple_logical_exp_cs`
	`logical_exp_cs = {simple_logical} simple_logical_exp_cs#chain`
	`simple_logical_exp_cs = {simple_logical} simple_logical_exp_cs simple_log relational_exp_cs`
	`simple_logical_exp_cs = {relational}<OclExpressionAS> relational_exp_cs#chain`
	`relational_exp_cs = {relational} relational_exp_cs relational_operator additive_exp_cs`
	`relational_exp_cs = {additive}<OclExpressionAS> additive_exp_cs#chain`
	`relational_operator = {relop} relop #chain| {equals} "="#chain`
	`additive_exp_cs = {additive} additive_exp_cs aop multiplicative_exp_cs`
	`additive_exp_cs = {multiplicative} multiplicative_exp_cs#chain`
	`multiplicative_exp_cs = {multiplicative} multiplicative_exp_cs mop unary_exp_cs`
	`multiplicative_exp_cs = {unary}<OclExpressionAS> unary_exp_cs#chain`
	`unary_exp_cs = {unary} unary_operator postfix_exp_cs`
	`unary_exp_cs = {postfix} postfix_exp_cs#chain`
	`unary_operator = {aop} aop #chain| {not} "not"#chain`

Bemerkung	Die erste Regel der originalen Grammatik steht für die mathematischen und logischen Operationen. Durch diese Regel kann es aber zu Mehrdeutigkeiten kommen und die Prioritäten werden nicht berücksichtigt. Aus diesem Grund wurden die arithmetischen Ausdrücke in der linksrekursiven Darstellung aufgebaut (siehe Abschnitt 2.2.2 auf Seite 25). Die zweite Regel der originalen Grammatik steht für die unären mathematischen Ausdrücke. Diese finden sich in den letzten drei Regeln der angepassten Grammatik wieder.

OCL-Spez.	`[B] OperationCallExpCS ::= OclExpressionCS "->" simpleNameCS "(" argumentsCS? ")"`
SableCC-Spez.	`postfix_exp_cs = {arrowright_operation_call} postfix_exp_cs "->" collection_operation` `"(" actual_parameter_list_cs? ")"`
Bemerkung	Die Regel der originalen Grammatik steht für die Anwendung einer Operation auf eine Menge von Objekten. Im Prinzip wurde die Regel in der angepassten Grammatik übernommen. Allerdings kann nicht jeder OCL-Ausdruck als Quellausdruck in Frage kommen (vgl. Nichtterminal `OclExpressionCS` in der originalen Grammatik), da es auch hier wieder zu Mehrdeutigkeiten kommen würde (siehe Beschreibung zum Nichtterminal `IteratorExpCS` der originalen Grammatik).

OCL-Spez.	`[C] OperationCallExpCS ::= OclExpressionCS "." simpleNameCS "(" argumentsCS? ")"` `[E] OperationCallExpCS ::= OclExpressionCS "." simpleNameCS isMarkedPreCS "(" argumentsCS? ")"`
SableCC-Spez.	`postfix_exp_cs = {dot_operation_call} postfix_exp_cs "." identifier_cs atpre?` `"(" actual_parameter_list_cs? ")"`
Bemerkung	Die beiden Regeln der originalen Grammatik bezeichnen einen normalen Operationsaufruf. Beide Regeln finden sich in der einen Regel der angepassten Grammatik wieder. Der Unterschied zwischen den Regeln der originalen Grammatik besteht in dem Zusatz `@pre`. Dieser wird in der Regel der angepassten Grammatik als optional gekennzeichnet.

OCL-Spez.	`[D] OperationCallExpCS ::= simpleNameCS "(" argumentsCS? ")"`
	`[F] OperationCallExpCS ::= simpleNameCS isMarkedPreCS "(" argumentsCS? ")"`
	`[G] OperationCallExpCS ::= pathNameCS "(" argumentsCS? ")"`

SableCC-Spez.	`property_call_exp_cs = {parameter} path_name_cs atpre? "(" actual_parameter_list_cs? ")"`
Bemerkung	Die ersten beiden Regeln der originalen Grammatik bezeichnen einen Operationsaufruf ohne Quellausdruck. Beide Regeln werden in einer Regel der angepassten Grammatik zusammengefasst. Der Ausdruck `@pre` ist als optional gekennzeichnet. Die dritte Regel der angepassten Grammatik bezeichnet einen statischen Operationsaufruf. Auch dieser wird auf die Regel der angepassten Grammatik abgebildet.

OCL-Spez.	`[A] AttributeCallExpCS ::= OclExpressionCS "." simpleNameCS isMarkedPreCS?`
SableCC-Spez.	`postfix_exp_cs = {dot_property_call} postfix_exp_cs "." identifier_cs atpre?`
Bemerkung	Die Regel der originalen Grammatik wurde so übernommen. Allerdings kann der Quellausdruck nicht jeder beliebige OCL-Ausdruck sein (vgl. Nichtterminal `OclExpressionCS` der originalen Regel), da es auch hier wieder zu Mehrdeutigkeiten kommen kann (siehe Nichtterminal `IteratorExpCS` der originalen Grammatik). Das Nichtterminal `simpleNameCS` der originalen Regel wurde zum Nichtterminal `identifier_cs` der angepassten Regel. Dieses Nichtterminal deckt jeden Namen ab.

OCL-Spez.	`[B] AttributeCallExpCS ::= simpleNameCS isMarkedPreCS?` `[C] AttributeCallExpCS ::= pathNameCS`
SableCC-Spez.	`property_call_exp_cs = {simple} path_name_cs atpre?`
Bemerkung	Beide Regeln der originalen Grammatik finden sich in der einen Regel der angepassten Grammatik wieder. Während der semantischen Analyse muss geprüft werden, ob ein `@pre`-Ausdruck nach einem statischen Attributaufruf geschrieben wurde. Wenn ja, handelt es sich dabei um einen Fehler.

OCL-Spez.	`[A] NavigationCallExpCS ::= AssociationEndCallExpCS` `[B] NavigationCallExpCS ::= AssociationClassCallExpCS`
SableCC-Spez.	keine Entsprechung
Bemerkung	Diese Weiterleitungsregeln finden sich nicht in der angepassten Grammatik wieder.

OCL-Spez.	`[A] AssociationEndCallExpCS ::= OclExpressionCS "." simpleNameCS ( "[" argumentsCS "]" )?` `isMarkedPreCS?`

SableCC-Spez.	`postfix_exp_cs = {dot_property_assoc_call} postfix_exp_cs "." identifier_cs`
	`"[" actual_parameter_list_cs "]" atpre?`
	
	`postfix_exp_cs = {dot_property_call} postfix_exp_cs "." id"entifier_cs atpre?`

Bemerkung	Die beiden Regeln in der originalen Grammatik bezeichnen den Zugriff auf ein Assoziationsende. Im Pivotmodell wird nicht zwischen einem normalen Attribut und einem Assoziationsende unterschieden. Daher wird die Regel ohne Qualifizierung (in der Regel der originalen Grammatik sind die Symbole `"[" argumentsCS "]"` optional) auf die Regel der normalen Attribute abgebildet (zweite Regel der angepassten Grammatik). Als Quellausdruck kann nicht jeder OCL-Ausdruck in Frage kommen, weil es sonst zu Mehrdeutigkeiten kommen würde (siehe Nichtterminal `IteratorExpCS` der originalen Grammatik). Das Nichtterminal `identifier_cs` bildet auf alle Namen wie `simpleNameCS` der originalen Grammatik ab.

OCL-Spez.	`[B] AssociationEndCallExpCS ::= simpleName ( "[" argumentsCS "]" )? isMarkedPreCS?`
SableCC-Spez.	`property_call_exp_cs = {association} path_name_cs "[" actual_parameter_list_cs "]" atpre?`
Bemerkung	Die Regel der originalen Grammatik wird in der angepassten Grammatik übernommen.

OCL-Spez.	`[A] AssociationClassCallExpCS ::= OclExpressionCS "." simpleNameCS ( "[" argumentsCS "]" )?`
	`isMarkedPreCS?`
	
	`[B] AssociationClassCallExpCS ::= simpleNameCS ( "[" argumentsCS "]" )? isMarkedPreCS?`

SableCC-Spez.	keine Entsprechung
Bemerkung	Im Pivotmodell gibt es keine Assoziationsklassen. Folglich müssen diese Regeln nicht abgebildet werden.

OCL-Spez.	`isMarkedPreCS ::= "@" "pre"`
SableCC-Spez.	keine Entsprechung
Bemerkung	Dieses Nichtterminal der originalen Grammatik wird in der angepassten Grammatik durch das Morphem `atPre` repräsentiert.

OCL-Spez.	`argumentsCS[1] ::= OclExpressionCS ( "," argumentsCS[2] )?`

SableCC-Spez.	`actual_parameter_list_cs = actual_parameter_list_element_cs actual_parameter_enum_cs*`
	`actual_parameter_enum_cs = "," actual_parameter_list_element_cs#chain`
	`actual_parameter_list_element_cs = {expression} ocl_expression_cs#chain`
	`actual_parameter_list_element_cs = {formal_parameter} formal_parameter_cs#chain`

Bemerkung	Die Regel der originalen Grammatik wurde auf vier Regel der angepassten Grammatik verteilt.

OCL-Spez.	`LetExpCS ::= "let" VariableDeclarationCS LetExpSubCS`
	`[A] LetExpSubCS[1] ::= "," VariableDeclarationCS LetExpSubCS[2]`
	`[B] LetExpSubCS ::= "in" OclExpressionCS`

SableCC-Spez.	`let_exp_cs = "let" initialized_variable_list_cs "in" ocl_expression_cs`
Bemerkung	Die drei Regeln der originalen Grammatik wurden in der Regel der angepassten Grammatik zusammengefasst. Im Gegensatz zu den originalen Regeln, lässt die Regel der angepassten Grammatik lediglich Variablenausdrücke mit initialem OCL-Ausdruck zu.

OCL-Spez.	`[A] OclMessageExpCS ::= OclExpressionCS "^^" simpleNameCS "(" OclMessageArgumentsCS? ")"` `[B] OclMessageExpCS ::= OclExpressionCS "^" simpleNameCS "(" OclMessageArgumentsCS? ")"`

SableCC-Spez.	`postfix_exp_cs = {msg} postfix_exp_cs msg_operator_cs signal_spec_exp_cs`
	`msg_operator_cs = {caret}"^" | {dblcaret}"^^"`
	`signal_spec_exp_cs = simple_name "(" message_argument_list_cs? ")"`

Bemerkung	Die Regeln der originalen Grammatik finden sich in den drei angegebenen Regeln der angepassten Grammatik wieder. Zu beachten ist, dass die Nachrichtenausdrücke im OCL-Parser nicht ausgewertet werden, da das Pivotmodell keine Nachrichten unterstützt.

OCL-Spez.	`OclMessageArgumentsCS[1] ::= OclMessageArgCS ( "," OclMessageArgumentsCS[2] )?`
SableCC-Spez.	`message_argument_list_cs = message_arg_cs message_arg_enum_cs*` `message_arg_enum_cs = "," message_arg_cs`
Bemerkung	Die Regel der originalen Grammatik findet sich in den angegebenen Regeln der angepassten Grammatik wieder.

OCL-Spez.	`[A] OclMessageArgCS ::= "?" ( ":" typeCS )?` `[B] OclMessageArgCS ::= OclExpressionCS`
SableCC-Spez.	`message_arg_cs = {questionmark_parameter} "?" formal_parameter_type_specifier?` `message_arg_cs = {expression_parameter} ocl_expression_cs`
Bemerkung	Die beiden Regeln der originalen Grammatik wurden in der angepassten Grammatik übernommen.

OCL-Spez.	`IfExpCS ::= "if" OclExpression[1] "then" OclExpression[2] "else OclExpression[3] "endif"`
SableCC-Spez.	`if_exp_cs = "if" ocl_expression_cs "then" ocl_expression_cs "else" ocl_expression_cs "endif"`
Bemerkung	Die Regel wurde so übernommen.

B.2 Neu hinzugefügte Regeln

In diesem Abschnitt werden die Regeln der angepassten Grammatik aufgeführt, die keine Entsprechung in der originalen Grammatik aus [OCL2] besitzen.

Regel	`ocl_file = packaged_constraint_list_cs+`
Bemerkung	Diese Regel vereint mehrere Paketdeklarationen.

Regel	`identifier_cs = {simple} simple_name#chain`
	`identifier_cs = {ocl_op_name} ocl_op_name#chain`
	`identifier_cs = collection_type_identifier_cs#chain`

Bemerkung	Das Nichtterminal `identifier_cs` steht für alle Identifizierer. Es kann sich dabei auch um ein Schlüsselwort handeln, denn ein Nutzer kann ein Attribut oder eine Operation wie eines der Schlüsselwörter benennen.

Regel	`postfix_exp_cs = {literal} literal_exp_cs#chain`
	`postfix_exp_cs = {parens} "(" ocl_expression_cs ")"#chain`
	`postfix_exp_cs = {if} if_exp_cs#chain`
	`postfix_exp_cs = {property} property_call_exp_cs#chain`

Bemerkung	Diese Regeln sind ein wenig vergleichbar mit der Definition des Nichtterminals `OclExpressionCS` aus der originalen Grammatik. Diese Regeln ergeben sich durch die Einführung der arithmetischen und logischen Ausdrücke. Im Prinzip reihen sich die Literalausdrücke, `if`-Ausdrücke und die anderen in die mathematischen und logischen Ausdrücke ein, sie besitzen die höchste Priorität, da sie "ganz unten" angesiedelt sind.

C Abbildung der Umgebung auf die OCL-Parser-Implementierung

In diesem Anhang wird die spezifizierte Umgebung der OCL-Spezifikation auf den Seite 91 bis 93 auf die Umgebung abgebildet, die im OCL-Parser Verwendung findet. Im ersten Abschnitt werden alle Methoden vorgestellt, die Bestandteil der originalen Umgebung sind, während im zweiten Abschnitt neu hinzugefügte Methoden erläutert werden.

C.1 Abbildung der originalen Umgebung an den OCL-Parser

Umgebung der OCL-Spez.	`lookupLocal(name :String): NamedElement`
Umgebung des OCL-Parsers	nicht vorhanden
Bemerkung	`lookupLocal` wird nicht benötigt, weil immer rekursiv in den Elternumgebungen gesucht wird.

Umgebung der OCL-Spez.	`lookup(name :String): ModelElement`

Umgebung des OCL-Parsers	`lookupType(List<String> pathName): Type`
	`lookupProperty(List<String> pathName): Property`
	`lookupOperation(String name, List<Type> params): Operation`
	`lookupOperation(List<String> pathName, List<Type> parameterTypes): Operation`

Bemerkung	Während der semantischen Prüfung ist es klar, was für ein Element gesucht werden soll (ob `Type`, `Property` oder `Operation`). Diese Information wird mit diesen vier Methoden weitergegeben. Zudem ist der Rückgabetyp wichtig.

Umgebung der OCL-Spez.	`lookupPathName(names: Sequence(String)): ModelElement`
Umgebung des OCL-Parsers	nicht vorhanden
Bemerkung	Ein `pathName` kann eine Klasse (bzw. *Type* im PivotModell), eine Operation, ein Attribut (bzw. *Property* im PivotModell) oder einen Namensraum bezeichnen. Die Suche wird über die entsprechenden `lookup`-Methoden abgedeckt.

Umgebung der OCL-Spez.	`addElement(name : String, elem: ModelElement, imp : Boolean): Environment`

Umgebung des OCL-Parsers	`addVariable(var : Variable): boolean`
	`addImplicitVariable(var : Variable): boolean`
	`setNamespace(ns : Namespace)`

Bemerkung	Die Methode `addElement` wurde durch die drei angegebenen Methoden ersetzt. Während der semantischen Analyse ist es immer ersichtlich, ob der Umgebung eine explizite Variable (vgl. Methode `addVariable`) oder eine implizite Variable (vgl. Methode `addImplicitVariable`) hinzugefügt werden soll. Diese beiden Methoden ersetzen das Attribut `imp` der originalen Methode. Die Methode `setNamespace` wird verwendet, um den Namensraum der Umgebung bekannt zu machen, der vom Nutzer vorgegeben wird (siehe Schlüsselwort `package`).

Umgebung der OCL-Spez.	`addEnvironment(env : Environment): Environment`
Umgebung des OCL-Parsers	nicht vorhanden
Bemerkung	Die Methode `addEnvironment` wird während der semantischen Prüfung nicht benötigt. Eine Umgebung besitzt einen Verweis auf eine Vorgängerumgebung, sofern es sich nicht um die Wurzelumgebung handelt. Durch diese Konstruktion kann nach einem Element rekursiv gesucht werden.

Umgebung der OCL-Spez.	`addNamespace(ns : Namespace): Environment`
Umgebung des OCL-Parsers	nicht vorhanden
Bemerkung	Während der semantischen Analyse muss genau ein Namensraum festgelegt werden. Dieser wird mit der Methode `setNamespace` gesetzt, so dass die Methode `addNamespace` überflüssig wird.

Umgebung der OCL-Spez.	`nestedEnvironment(): Environment`
Umgebung des OCL-Parsers	`nestedEnvironment(): Environment`
Bemerkung	Diese Methode erzeugt eine neue Umgebung und setzt den Vorgänger dieser Umgebung auf die Instanz, deren Methodenaufruf zu der neuen Umgebung geführt hat.

Umgebung der OCL-Spez.	`lookupImplicitAttribute(name : String): Attribute` `lookupImplicitAssociationEnd(name : Strng): AssociationEnd`
Umgebung des OCL-Parsers	`lookupImplicitProperty(name : String): Variable`
Bemerkung	Die Attribute (bzw. *Property* im PivotModell) und Assoziationsenden fallen im Pivotmodell zum Konstrukt *Property* zusammen. Daher wird nur eine Methode benötigt. Der Rückgabewert hat sich im Vergleich zur originalen Methodensignatur geändert. Im OCL-Parser wird eine Variable mit Namen und Typ zurückgegeben, da nur eine implizite Variable auf ein Attribut verweisen kann.

Umgebung der OCL-Spez.	`lookupImplicitOperation(name : String, params : Sequence(Classifier)): Operation`
Umgebung des OCL-Parsers	`lookupImplicitOperation(name : String, params : List<OclExpression>)): Operation`
Bemerkung	Sucht eine implizite Operation, das heißt, eine Operation wurde ohne Quellausdruck aufgerufen. Es wird mit dem Namen `name` und den Parametern `params` nach einer Operation gesucht, die ein Typ der impliziten Variablen oder die Variable `self` besitzen könnte.

Umgebung der OCL-Spez.	`lookupImplicitSourceForAttribute(name : String): NamedElement`
Umgebung des OCL-Parsers	nicht vorhanden
Bemerkung	Die Methode `lookupImplicitSourceForAttribute` wird nicht benötigt, da im Pivotmodell jede Eigenschaft ihren Quelltypen kennt (siehe Methode `Property::getSource()` im Paket `tudresden.ocl20.pivot.pivotmodel`).

C.2 Neu hinzugefügte Methoden

Hier werden Methoden der Umgebung aufgeführt, die nicht Bestandteil der originalen Umgebung aus [OCL2] sind.

Umgebung des OCL-Parsers	`setSepcialOclOperation(value : boolean)` `getSpecialOclOperation(): boolean`
Bemerkung	Die Parameter der Operationen `oclIsKindOf`, `oclIsTypeOf` und `oclAsType` müssen besonders behandelt werden. Wenn eine solche Operation aufgerufen wird, muss als Parameter genau eine Typbezeichnung angegeben werden. Mit diesen Methoden der Umgebung kann der semantischen Auswertung Kontextinformation mitgegeben werden (siehe auch Abschnitt 6.3.15 auf Seite 121).

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Literaturverzeichnis

[AHO] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullmann. Compilers Prinicples, Technique, & Tools, Second Edition. Pearson Addison Wesley, 2007

[ANS] Ansgar Konermann. Entwurf und prototypische Implementation eines OCL2.0-Parsers. Diplomarbeit, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, August 2003

[ANT] Ant Webseite. Webseite: http://ant.apache.org/ . Letzter Zugriff 2007.

[Antlr] Antlr Webseite. Webseite: http://www.antlr.org/ . Letzter Zugriff: 2007

[ASB] Alexander Asteroth, Christel Baier. Theoretische Informatik, Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen. Pearson Studium, 2002

[AspectJ] AspectJ Webseite. Webseite: http://www.eclipse.org/aspectj . Letzter Zugriff: 2007

[Beaver] Beaver Webseite. Adresse: http://beaver.sourceforge.net/ . Letzter Zugriff 2007

[BÖH] Oliver Böhm. Aspektorientierte Programmierung mit AspectJ5, Einsteigen in AspectJ und AOP. dpunkt Verlag, 2006

[Braeuer] Matthias Bräuer. Design and Prototypical Implementation of a Pivot Model as Exchange Format for Models and Metamodels in a QVT/OCL Development Environment. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, Mai 2007

[BRA] Ronny Brandt. Ein OCL-Interpreter für das Dresden OCL2 Toolkit basierend auf dem Pivotmodell. Diplomarbeit, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, November 2007

[CUP] CUP Webseite: http://www2.cs.tum.edu/projects/cup/ . Letzter Zugriff 2007.

[Dot] Graphviz Webseite. Webseite: http://www.graphviz.org . Letzter Zugriff: 2007

[Eclipse] Eclipse Webseite. Webseite: http://www.eclipse.org . Letzter Zugriff: 2007

[EMF] Eclipse Modeling Framework Webseite: http://www.eclipse.org/modeling/emf/. Letzter Zugriff 2007

[GAG] Étienne Gagnon. SableCC, An object-oriented compiler framework. Thesis, School of Computer Science McGill University, Montreal, 1998

[GOF] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley Verlag, Juli 2004

[GSCK] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent. Software Factories. Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004

[FIN] Frank Finger. Java-Implementierung der OCL-Basisbibliothek. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, Juli 1999

[IDL] OMG IDL Details. Webseite: http://www.omg.org/gettingstarted/omg_idl.htm/. Letzter Zugriff 2007

[JAST] Görel Hedin, Eva Magnusson. JastAdd - an aspect-orientied compiler construction system. Paper, Department of Computer Science, Lund University, Sweden, Mai 2002

[JAPAR] Parser-Empfehlung für JastAdd. Webseite: http://jastadd.cs.lth.se/web/manual/concepts.shtml#parser.

Letzter Zugriff: 2007

[JASTWeb] JastAdd Webseite. Webseite: http://jastadd.cs.lth.se/web . Letzter Zugriff: 2007

[JFlex] JFlex Webseite. Webseite: http://jflex.de/ . Letzter Zugriff: 2007

[JUnit] JUnit Websete. Webseite: http://www.junit.org/ . Letzter Zugriff: 2007

[KEN] KentOCL Webseite: http://www.cs.kent.ac.uk/projects/ocl/ . Letzter Zugriff 2007

[KFM] Kent Modeling Framework Webseite: http://www.cs.kent.ac.uk/projects/kmf/. Letzter Zugriff 2007

[Kon] Ansgar Konermann. The Parser Subsystem of the Dresden OCL2 Toolkit Design and Implementation. Internes Papier, Technische Universität Dresden, Lehrstuhl für Softwaretechnologie, 2005

[Lex] Lex Webseite. Webseite: http://dinosaur.compilertools.net/ . Letzter Zugriff: 2007

[LPG] LALR Parser Generator (LPG) Webseite: http://sourceforge.net/projects/lpg/. Letzter Zugriff 2007

[MDA] Object Management Group (OMG). MDA Guide 1.0.1. Version 1.0.1, 2003

[MDT] Modeling Development Tools Webseite: http://www.eclipse.org/modeling/mdt/?project=ocl . Letzter Zugriff 2007

[MOF] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification. Version 2.0, 2006

[NAO] Naomi - OCL Validation Framework. Webseite: https://sourceforge.net/projects/mocl/ . Letzter Zugriff 2007.

[OCL2] Object Management Group (OMG). Object Constraint Language. Version 2.0, 2006

[OMG] OMG Webseite. Webseite: http://www.omg.org/ . Letzter Zugriff: 2007

[Schmitz] Lothar Schmitz. Syntaxbasierte Programmierwerkzeuge. B.G. Teubner Stuttgart 1995

[SVEH] Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase. Modellgetriebene Softwareentwicklung, Techniken, Engineering, Management. dpunkt.verlag, 2007

[UML] Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.1.1, 2007

[War] Jos Warmer, Anneke Kleppe. The Object Constraint Langauge: Getting your models ready for MDA, Second Edition. Addision Wesley, 2003

[Wirth] Niklaus Wirth. Compilerbau: Eine Einführung. Teubner Verlag, 1986