Diplomarbeit

Ein Compiler für eine Spezifikationssprache für föderierte Datenbankschemata und deren Abhängigkeiten

Patrick Koehne

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

11. Januar 1999

Gutachter:

Dr. W. Hasselbring
Prof. Dr. E.-E. Doberkat

Vorwort

Am Lehrstuhl für Software-Technologie der Universität Dortmund wurde zeitgleich an zwei Diplomarbeiten im Themenbereich *föderierte Datenbanken* gearbeitet. Für beide Diplomarbeiten dienten Ergebnisse aus der Projektgruppe 290 [Pro97] als Grundlage, an der beide Diplomanden teilgenommen haben.

Die im Rahmen beider Arbeiten entwickelten Werkzeuge verfügen mit der in Kapitel 3 vorgestellten Spezifikationssprache über eine gemeinsame Schnittstelle. Auf Grund dieser Tatsache und dem zeitgleichen Entstehen beider Diplomarbeiten wurden einige Teile dieser Arbeit gemeinsam entwickelt und die dazugehörigen Dokumentationen zusammen verfaßt.

Inhaltsverzeichnis

I Die Grundlagen (Patrick Koehne, Sven Gerding) 1

1 Einleitung 3

1.1 Hintergrund . . . 3

1.2 Eigenschaften eines FDBS . . . 3

1.3 Schemaintegration . . . 4

2 Der Föderierungsgraph 7

2.1 Einleitung . . . 7

2.2 Die Schema-Architektur . . . 7

2.3 Anforderungen . . . 9

2.4 Das Meta-Datenmodell . . . 10

2.4.1 Die Klassenhierarchie . . . 10

2.4.2 Die Konverterklassen . . . 11

3 Die Spezifikationssprache 13

3.1 Motivation . . . 13

3.2 Anforderungen an die Spezifikationssprache . . . 14

3.3 Entwurfsentscheidungen . . . 15

3.4 Beschreibung der Sprache . . . 20

3.4.1 Schemadefinition . . . 21

3.4.2 Definition von Abhängigkeiten . . . 22

3.4.3 Nesting/Unnesting . . . 22

3.4.4 Wertekonvertierung . . . 23

3.5 Ausblick . . . 24

II Der Compiler 25

4 Einleitung 27

5 Die Aufgabe des Compilers **28**
5.1 Die Übersetzung 28
5.2 Fehlermeldungen 29

6 Die Eingabesprache **30**
6.1 Allgemeine Beschreibung 30
6.2 Das Vokabular 31
6.3 Die Grammatik 32
6.3.1 Definitionsblock - Grammatik 33
6.3.2 Die Mappingblock - Grammatik 34
6.4 Beispiel 36

7 Die Zielstruktur **37**
7.1 Programmkopf 37
7.2 Der Spezifikationsteil 37
7.2.1 Schemadefinition 37
7.2.2 Klassendefinition 38
7.2.3 Pointerdefinition 39
7.2.4 Listendefinition 39
7.2.5 Attributdefinitionen 39
7.2.6 Inversdeklaration 40
7.3 Der Mappingteil 40
7.3.1 Schemamapping 40
7.3.2 Typemapping 40
7.3.3 Attributmapping 41
7.4 Kommentare 41

8 Aufbau des Compilers mit Eli **42**
8.1 Teilprobleme 42
8.2 Wie ein Compiler spezifiziert wird 45
8.3 Der eigentliche Compiler 46

9 Lexikalische Analyse **48**

10 Syntaktische Analyse **50**

11 Bereichsanalyse **52**

12 Typ-Analyse **54**

13 Die Ausgabemuster **55**

14 Generierung der Ausgabe **58**
14.1 Die Ausgabe . . . 58
14.2 Generierung ausgabefähiger Attribute . . . 59
14.3 Berechnungen für den Definitionsblock . . . 59
14.4 Berechnungen für den Mappingblock . . . 62
14.4.1 Mapping-From . . . 63
14.4.2 Map-From-Nest . . . 65
14.4.3 Mapping-To . . . 66
14.4.4 Map-To-Nest . . . 66
14.5 Die Ausgabe . . . 68

15 Das Laufzeitsystem in O_2 **70**
15.1 Die Weiterverarbeitung der Compilerausgabe . . . 70
15.2 Die Portierung der Graphklassen . . . 70
15.2.1 Vorbereiten der Graphklassen . . . 71
15.2.2 Import der Klassen in O_2 . . . 72
15.3 Benutzung der Graphklassen . . . 72

16 Zusammenfassung **73**
16.1 Rückblick . . . 73
16.1.1 Das Compilerbauwerkzeug ELI . . . 73
16.1.2 Die Datenbank O_2 . . . 74
16.2 Ausblick . . . 74

III Evaluation (Sven Gerding, Patrick Koehne) **77**

17 Die Testumgebung **79**
17.1 Überblick . . . 79
17.2 Das Szenario . . . 79
17.3 Ablauf . . . 80
17.3.1 Generierung der Datenbanktabellen . . . 80
17.3.2 Umsetzung in Komponentenschemata . . . 81
17.3.3 Aufbau des Föderierungsgraphen . . . 81

17.3.4 Compilation der Spezifikation 83
17.3.5 Aufbau des Testsystems 83
17.3.6 Test des Föderierungssystems 84
17.4 Ergebnisse 85

A Grammatik der Spezifikationssprache 86
A.1 Vokabular 87
A.2 Grammatik 88

B Erläuterungen zur lexikalischen Analyse 91
B.1 Der Quelltext 91
B.2 Der Zwischencode 91
B.3 Das Scannen 92
B.4 Identifier Table 93
B.5 Die Spezifikationsdatei für die lexikalische Analyse 93

C Erläuterungen zur syntaktischen Analyse 95
C.1 `name` in verschiedenen Kontexten 95
C.2 Die Spezifikationsdatei für die syntaktische Analyse 96
C.3 Die Grammatik in Eli-Notation 96

D Erläuterungen zu den Ausgabemustern 99
D.1 Programmkopf 99
D.2 Der Definitionsteil 99
D.2.1 Schemadefinition 101
D.2.2 Klassendefinition 101
D.2.3 Inverse-Attribute-Definition 103
D.2.4 Attribut-Definition 103
D.2.5 Pointer-, Listen- und Generic-Definition 103
D.2.6 Zusätzlich benötigte Muster 103
D.3 Der Mappingteil 105
D.3.1 Pre- und Post-Verbindungen 105
D.3.2 Parameterlisten- und Konverterdefinition 105
D.3.3 Weitere benötigte Muster 105
D.4 Spezifikationsdatei für die Ausgabemuster 108

E Erläuterungen zu der Generierung der Ausgabe **109**

E.1 Generierung ausgabefähiger Attribute . . . 109

E.2 Berechnungen für den Definitionsblock . . . 111

E.3 Berechnungen für den Mappingblock . . . 113

E.3.1 Mapping-From . . . 114

E.3.2 Map-From-Nest . . . 115

E.3.3 Map-To-Nest . . . 116

E.4 Spezifikationsdateien für die Berechnungen . . . 118

E.5 Die Ausgabe . . . 118

E.5.1 Programmkopf . . . 118

E.5.2 Definitionsblock . . . 119

E.5.3 Mappingblock . . . 121

E.6 Die Spezifikationsdatei für die Ausgabe . . . 123

F Erläuterungen zum Laufzeitsystem **125**

F.1 Konfigurationsdatei für O_2Makegen . . . 125

G Das Testsystem **126**

G.1 Realisierung der Grapherzeugung . . . 126

G.1.1 *C*++-Quellcode für die Graphgenerierung . . . 127

G.2 Realisierung der Konvertierungsfuktionen . . . 129

G.2.1 Die Klasse CConcatConvert . . . 130

G.2.2 Die Klasse CEuroToDmConvert . . . 130

G.3 Die Graphalgorithmen . . . 131

G.3.1 Beispielmethoden der Klasse `CSpecialAlgorithms` . . . 133

G.4 Der Föderierungskern . . . 134

G.4.1 Quellcode des Föderierungskerns . . . 134

G.5 Agenten der Komponentendatenbanken . . . 136

G.5.1 bookdb . . . 136

G.5.2 inagent . . . 138

Teil I

Die Grundlagen

Kapitel 1

Einleitung

1.1 Hintergrund

Im Laufe der Jahre hat die EDV Einzug in fast alle Bereiche von Unternehmen gehalten, seien sie öffentlich oder privat, der Industrie oder dem Dienstleistungsgewerbe zugehörend. Größtenteils fanden dabei auf die Bedürfnisse des jeweiligen Unternehmenszweiges bzw. der Abteilung zugeschnittene Datenbankanwendungen Verwendung, was zur Folge hat, daß oftmals Daten (z.B. der Kundenstamm) redundant in mehreren heterogenen Umgebungen gehalten werden.

Heutzutage stellt sich das Problem, daß diese vielen verschiedenen Insellösungen aus organisatorischen oder Kostengründen zusammengefaßt werden sollen. So bedeutet die mehrmalige Erfassung gleicher Daten z.B. einen höheren Arbeitsaufwand, was zu größeren Kosten führt. Gleichzeitig stellt dies auch eine Fehlerquelle dar, da sich möglicherweise Inkonsistenzen im Datenbestand ergeben können.

Eine Möglichkeit wäre sicher, die alten Datenbanken und damit die Anwendungen durch ein neues System zu ersetzen, jedoch ist dies mit sehr hohen Kosten verbunden, da nicht nur die Datenbank und die Anwendungen neu angeschafft oder entwickelt, sondern auch die Mitarbeiter in das neue System eingearbeitet werden müssen.

Als Alternative hierzu bietet sich eine Föderation von Datenbanken an. Das bedeutet, daß die eingesetzen Datenbanken samt Datenbestand erhalten bleiben, ebenso die verschiedenen Anwendungen. Die Daten werden nach wie vor redundant, doch durch eine vom Föderierungssystem kontrollierte Replikation konsistent gehalten.

Ein weiterer Vorteil ergibt sich daraus, daß über das Föderierungssystem ein globaler Zugriff auf die an der Föderation beteiligten Datenbanken erfolgen kann. So ist die Möglichkeit geschaffen, Applikationen einzusetzten, die direkt auf dem föderierten Schema aufsetzen.

1.2 Eigenschaften eines FDBS

Wie schon zuvor erwähnt dient ein föderiertes Datenbanksystem, kurz FDBS, unter anderem der kontrollierten Replikation von Daten in den an der Föderation beteiligten Datenbanken. Dieser Abgleich von Daten wird von sogenannten Agenten überwacht, die direkt auf der Datenbank arbeiten. Der Benutzer muß (soll) von diesen Programmen nichts wissen - er arbeitet wie gewohnt mit seiner Applikation unter weitgehender Wahrung deren lokaler Autonomie. Der Zugriff auf die Daten erfolgt

voll transparent, die Heterogenität der Systeme und insbesondere die semantische Heterogenität der Daten bleibt verborgen.

Abbildung 1.1 zeigt beispielhaft eine Föderation von drei Datenbanken. In dieser Abbildung setzen jedoch keine externen Applikationen auf dem FDBS auf. Für die Föderation relevanten Informationen, wie z.B. Abhängigkeiten zwischen Schemata, werden in einer gesonderten Datenbank persistent gehalten. Die Agenten dienen der Kapselung der Komponentendatenbanken und sind für die Translation der lokalen in Komponentenschemata und umgekehrt zuständig.

1.3 Schemaintegration

In einem Föderierungssystem werden die Abhängigkeiten zwischen den beteiligten Datenbanken beschrieben. Um dies zu bewerkstelligen müssen die einzelnen Datenbankschemata abgeglichen werden.

Dazu legt man sich zunächst auf ein einheitliches kanonisches Datenmodell (z.B. objektorientiert) fest, in das die lokalen DB-Schemata transformiert werden [SL90]. Dem eigentlichen Integrationsprozeß geht also der Prozeß der Datenmodell-Translation voraus, der zu bezüglich des Datenmodells homogenen Schemata führt. Zu beachten ist, daß in der Regel nicht nur die Datenmodelle der jeweiligen lokalen DB-Schemata heterogen sind, sondern auch und insbesondere deren Struktur, was auf den ursprünglichen Entwurfsprozeß zurückzuführen ist. Schließlich existiert mehr als eine Darstellung eines realen Objektes. Erst nachdem die Schemata auch bezüglich ihrer Struktur abgeglichen, also homogen sind, können Abhängigkeiten zwischen ihnen spezifiziert werden.

Mit Hilfe der Schemaintegration ist es in einem föderierten Datenbanksystem möglich diese Unterschiede zu definieren, sie zu begleichen und somit trotzdem eine Föderation zu ermöglichen. Die Spezifikation solcher Abhängigkeiten wird in einer geeigneten Datenstruktur gespeichert, die von Hand angelegt werden muß. Da zum momentanen Zeitpunkt keine standardisierte Datenstruktur für diesen Verwendungszweck existiert, sind auch keine Werkzeuge vorhanden, mit deren Hilfe eine Teilautomatisierung dieses Vorganges möglich wäre. Um die Konstruktion einer solchen Datenstruktur nun übersichtlich und dadurch auch leichter als von Hand zu gestalten, wäre es wünschenswert diese Datenstruktur nicht direkt in einer konkreten Programmiersprache zu implementieren, sondern sie zunächst problemorientiert zu spezifizieren oder interaktiv mit Hilfe eines geeigneten Editors zu konstruieren. Dadurch erreicht man bei der Pflege solcher Datenstrukturen eine höhere Flexibilität bei Änderungen. Des weiteren ist es möglich eine formale Spezifikation automatisch weiterzuverarbeiten, z.B. graphisch anschaulich zu machen oder mit Hilfe eines Compilers die eigentliche Datenstruktur zu erzeugen. Außerdem wird der Entwickler in seinem Konstruktionsprozeß geführt und vermeidet somit Fehler.

Im Rahmen dieser Diplomarbeit sollen eine Spezifikationssprache und ein Compiler entstehen, der aus einer gegebenen Spezifikation gemäß der Sprache aus Kapitel 3 einen *C++*- Quellcode erzeugt, der wiederum die für das Föderierungssystem notwendige Datenstruktur in einer O_2 - Datenbank anlegt. Eine genauere Beschreibung dieser Datenstruktur findet sich im nachfolgenden Kapitel 2. Abbildung 1.2 zeigt die Architektur dieses Systems.

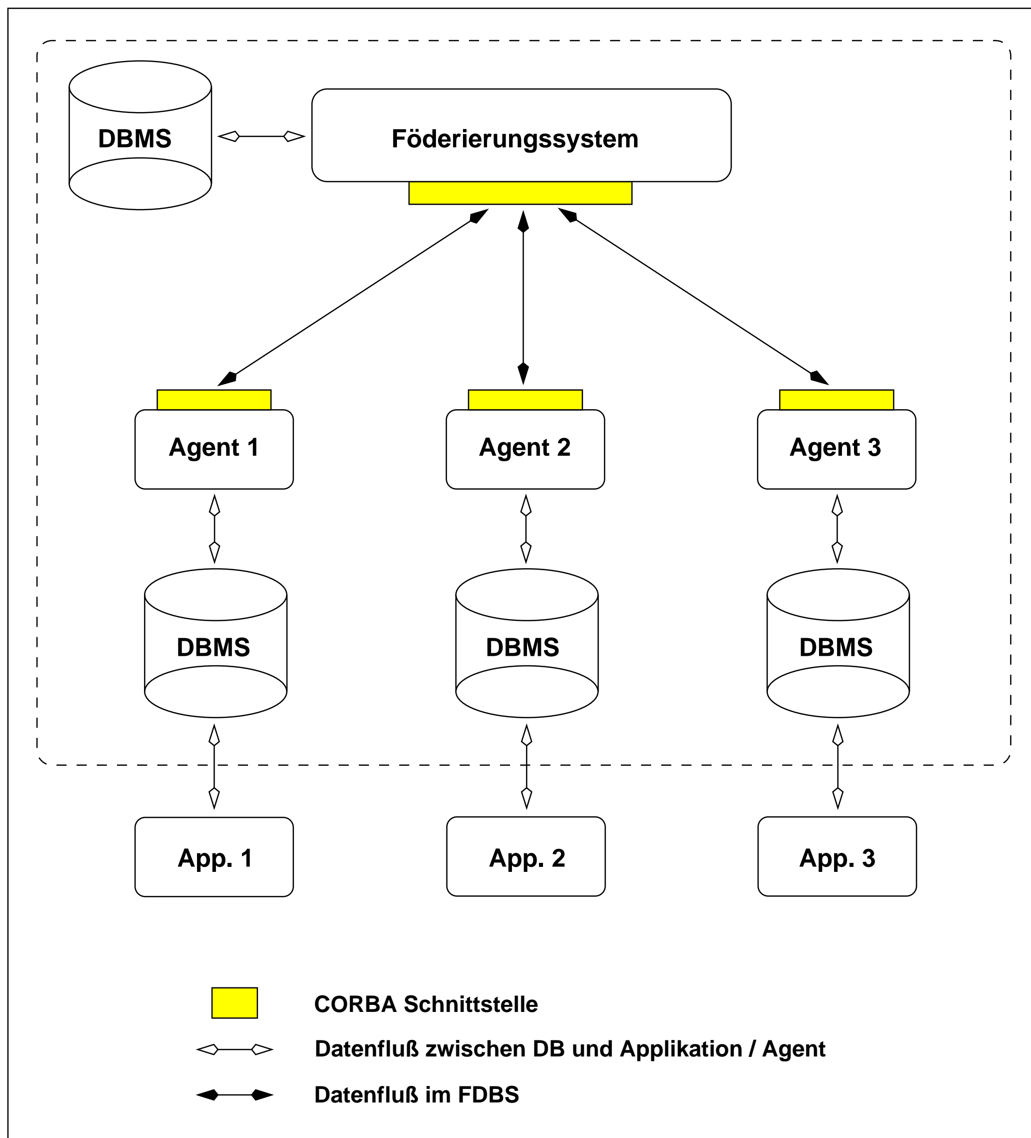

Abbildung 1.1: Föderation dreier Datenbanken.

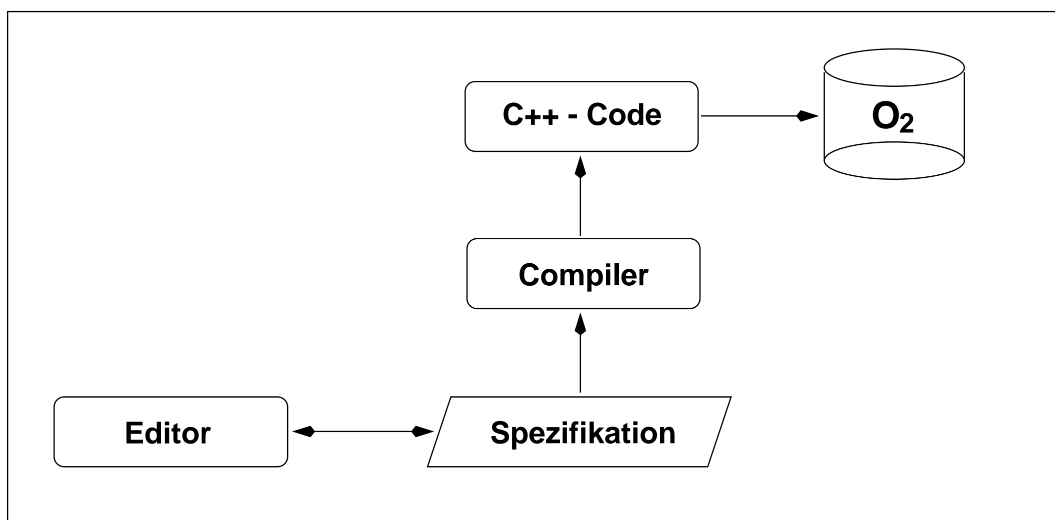

Abbildung 1.2: Systemarchitektur der Spezifikationssprache und des Compilers (Datenflußdiagramm).

Kapitel 2

Der Föderierungsgraph

2.1 Einleitung

Der Föderierungsgraph ist eine wesentliche Komponente eines föderierten Datenbanksystems. Er ist so aufgebaut, daß die unterschiedlichsten Datenbanken föderiert werden können. Zu diesem Zweck müssen Informationen über die lokalen Datenbankschemata vorliegen und in geeigneter Form vom Föderierungssystem gespeichert werden.

Der Föderierungsgraph ist ein Meta-Datenmodell (Abschnitt 2.4), in dem derartige Schemabeschreibungen definiert und gespeichert werden können. Auf einem solchen Graphen können dann andere Komponenten des Föderierungssystems arbeiten und die für eine Föderation notwendige Funktionalität zur Verfügung stellen.

2.2 Die Schema-Architektur

Die Föderierung der lokalen Datenbanken erfolgt in Anlehnung an die Sheth-Larson-Referenzarchitektur [SL90]. In Abbildung 2.1 ist der Aufbau dieser Architektur dargestellt. Die in dieser Arbeit verwendete Schemaarchitektur weicht von der oben erwähnten Sheth-Larson-Referenzarchitektur ab und ist gezielt für die Replikation von Datenbeständen entwickelt worden [Has97]. Abbildung 2.2 zeigt einen solchen Graphen in einer vereinfachten Form für n zu föderierende Datenbanken. Die unten aufgeführten Punkte umreißen kurz diesen Vorgang.

- Für jede angeschlossene lokale Datenbank existiert ein Komponentenschema, in dem das lokale Datenbankschema beschrieben wird. Diese Beschreibung erfolgt bereits in dem gewählten kanonischen Datenmodell, in dieser Arbeit einem objektorientierten Datenmodell.

- Zu jedem Komponentenschema werden nun Import- und Exportschemata angelegt. Diese beschreiben welche Teile der Komponentenschemata föderiert werden.

- Ein föderiertes Schema speichert letztendlich die Abhängigkeiten zwischen den Import- und Exportschemata der verschiedenen lokalen Datenbanken. Dies entspricht der Definition, welche Daten wohin repliziert werden sollen.

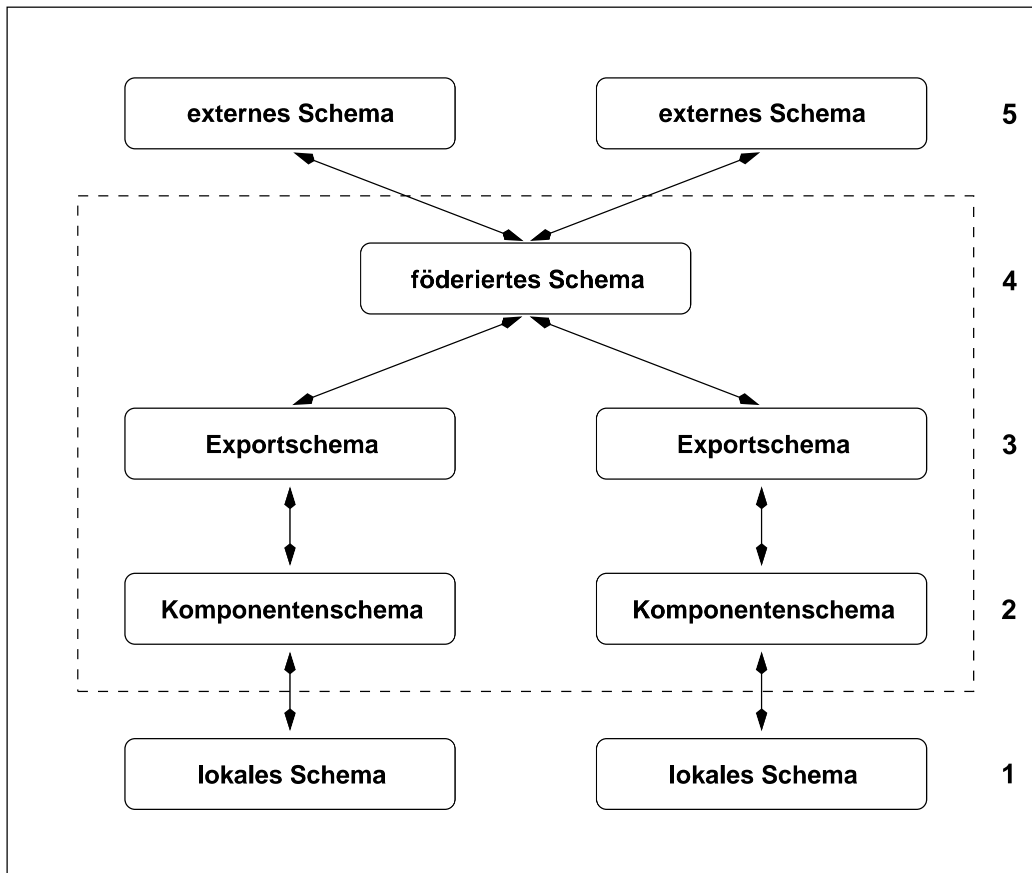

Abbildung 2.1: Die 5-Ebenen-Architektur nach Sheth-Larson.

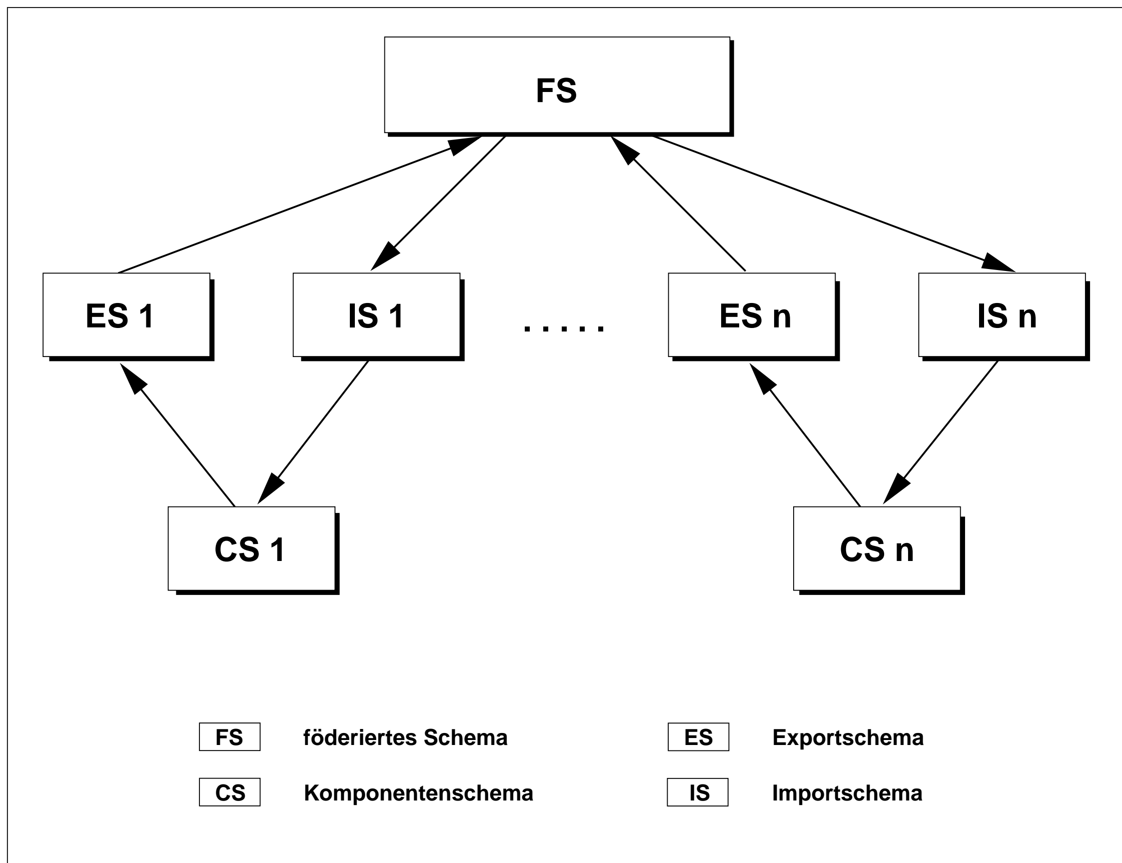

Abbildung 2.2: Vereinfachter Graph für n angeschlossene Datenbanken

Die hier angesprochenen Schematypen können mit Hilfe des Meta-Datenmodells definiert und die geforderten Abhängigkeiten zwischen den Schemata festgelegt werden. Hieraus ergibt sich dann der Föderierungsgraph.

Der Föderierungsgraph ist somit lediglich eine statische Datenstruktur, die dem Föderierungssystem als Grundlage dient. Die für das Föderierungssystem notwendige Funktionalität liefern Algorithmen, die sich der im Graphen definierten Informationen bedienen.

Im Rahmen dieser Diplomarbeit wird die Einschränkung gemacht, daß die zu generierenden Graphen nur dazu benutzt werden, eine konsistente Replikation von Daten zwischen verschiedenen Datenbanken automatisiert zu ermöglichen und zu kontrollieren. Aus diesem Grund wird der Fall, daß es eventuell globale Applikationen geben kann, die auf dem föderierten Schema arbeiten wollen, nicht betrachtet. Dies ist der Grund warum die Definition von externen Schemata gemäß Sheth-Larson entfällt. Das Meta-Datenmodell würde aber im Prinzip solche externen Schemata zulassen, so daß eine spätere Erweiterung durchaus möglich ist.

2.3 Anforderungen

Bei der Auslegung eines Föderierungssystems ist zu beachten, daß unterschiedliche Datenbanksysteme und damit auch unterschiedliche Datenmodelle zusammengeführt werden müssen. Dies soll

jedoch nicht mehr Aufgabe des Föderierungsgraphen sein und auch nicht die des Kerns des Föderierungssystems. Die Umwandlung der Datenmodelle in das kanonische Datenmodell des Föderierungsgraphen findet in Adaptern statt, die direkt auf den Datenbanken arbeiten und die Verbindung zwischen Datenbanken und Föderierungskern herstellen.

Im Rahmen der Projektgruppe 290 (FOKIS) [Pro97] ist ein Föderierungssystem entwickelt worden, dessen Kern der Föderierungsgraph darstellt. Das in der Projektgruppe erarbeitete Datenmodell soll in dieser Diplomarbeit Verwendung finden und entsprechend erweitert werden. Einige Erweiterungen gegenüber der Projektgruppe werden sein:

- Der Graph wird zwar auch in *C++* implementiert werden, aber er wird mittels eines geeigneten Mechanismus in der objektorientierten Datenbank O_2 [O296a] angelegt und gespeichert.
- Datenmodifizierungen bei der Föderierung werden möglich sein, z.B. Umwandlungen von Einheiten wie Liter in Gallonen oder auch das Umformatieren von Attributen, wie z.B. unterschiedliche Datumsformate. Die dafür notwendigen Methoden können während des Integrationsprozesses bereits an geeigneter Stelle mittels *C++* - Code spezifiziert werden. Sie werden dann an geeigneter Stelle an die Attribute im Schema angehängt und können bei der Föderierung direkt aufgerufen werden.
- Nesting/Unnesting: Hierunter versteht sich die Möglichkeit eine Menge von Attributen zu einem Attribut zusammenzufassen oder auch mehrere Klassen in eine Klasse zu überführen und umgekehrt.

2.4 Das Meta-Datenmodell

Um eine Föderation unterschiedlicher Datenbanken zu ermöglichen, muß ein Föderierungssystem in der Lage sein, Informationen über lokale Datenbaken zu verwalten. Dazu wird ein *Meta-Datenmodell* benötigt, mit dessen Hilfe sich Beschreibungen der lokalen Datenbank-Schemata ablegen und Abhängigkeiten zwischen diesen spezifizieren lassen. Das hier verwendete Meta-Datenmodell wurde in der PG 290 entwickelt und lehnt sich an den ODMG-93-Standard [Cat96] an.

Klassen des Meta-Datenmodells sind *Metaklassen*. Sie stellen unterschiedliche Klassentypen dar. Instanzen dieser Metaklassen wiederum beschreiben konkrete Klassen von Datenbank-Schemata. Zwischen diesen konkreten Klassen lassen sich nun Abhängigkeiten spezifizieren, woraus schließlich eine Graphstruktur entsteht, die als Föderierungsgraph bezeichnet wird. Schemabeschreibungen stellen Knoten und die Abhängigkeiten zwischen diesen stellen Kanten dar.

2.4.1 Die Klassenhierarchie

Nachfolgend sind die Klassen des Meta-Datenmodells aufgeführt und kurz erläutert. Für eine detaillierte Beschreibung dieser Klassen sei auf [Pro97] verwiesen.

CGraph: Instanzen der Klasse CGraph dienen zur Aufnahme verschiedener Schemata, die zusammen einen Föderierungsgraphen bilden. *CGraph* bildet gewissermaßen den Einstiegspunkt in einen Föderierungsgraphen.

CSchema: Über diese Klasse werden föderierte Datenbankschemata modelliert. Objekte vom Typ *CSchema* können dabei verschiedene Objekte der Typen *CSimpleType*, *CCollectionType*, *CPointerType*, sowie *CComplexType* enthalten.

CType: Diese Klasse dient als abstrakte Basisklasse für *CSimpleType*, *CPointerType*, *CCollection-Type* und *CComplexType*.

CSimpleType: Zur Realisierung atomarer Typen, wie *integer*, *string*, *char*, etc. werden Instanzen der Klasse *CSimpleType* benutzt.

CPointerType: Zur Definition von Referenzen auf andere Typen dient *CPointerType*.

CCollectionType: Mittels dieser Klasse lassen sich Listen modellieren, wobei zu beachten ist, daß deren Elementtyp im selben Schema wie die Liste selbst liegen muß.

CComplexType: Mit Hilfe der Klasse *CComplexType* werden Klassen modelliert. Elemente der Klassen sind dabei Objekte vom Typ *CAttribute*.

CAttribute: Attribute bilden den Inhalt von Klassen. Instanzen von *CAttribute* werden in Instanzen der Klasse *CComplexType* eingetragen. Jedes Attribut hat einen Typ, d.h. es referenziert Objekte vom Typ *CSimpleType*, *CCollectionType*, *CPointerType* oder *CComplexType*.

Ein Föderierungsgraph besitzt Schemata. Schemata besitzen Typen, wobei einige Typen (*CComplex-Type*) Attribute beinhalten können. Schemata, Typen und Attribute bilden die Knoten eines Föderierungsgraphen. Kanten werden erzeugt, indem Objekte auf Schema-, Typ-, und Attributebene miteinander verbunden werden. Hierzu verfügen alle Klassen, bis auf *CGraph*, über Methoden, mit denen sich Abhängigkeiten definieren lassen. Des weiteren können über entsprechende Funktionen ihre Namen und Referenzen gesetzt und ausgelesen werden.

2.4.2 Die Konverterklassen

Für die Realisierung der Konvertierungs- und Nestingfunktionalität des Graphen müssen die in diesem Abschnitt vorgestellten *Metaklassen* erweitert werden.

Hierzu wird das Modell einer *abstrakten* Konverterklasse eingeführt. Diese Klasse *CConvert* definiert die Schnittstelle für alle späteren Konverterklassen. Um nun Konverterklassen zu implementieren und im Graphen verfügbar zu machen, werden konkrete Klassen aus der abstrakten Klasse CConvert abgeleitet. In diesen Klassen wird dann *Convert* als die einzige Methode implementiert, so daß bei der späteren Konvertierung in jedem Fall dieselbe Methode auszuführen ist, egal welche Berechnungen tatsächlich getätigt werden müssen. Eine Instanz der konkreten Konverterklasse wird später während der Generierung des Föderierungsgraphen an das zu verändernde Attribut angehängt. Für die Attribute, die in der Spezifikation ohne Konvertierungsmethoden angegeben werden, wird eine Instanz der vorgegebenen Klasse *CIdentifyConvert* generiert. Diese Methode liefert die Identität des Attributes zurück, so daß keine Konvertierung stattfindet.

Das Klassenmodell aus Abbildung 2.3 zeigt diesen Zusammenhang.

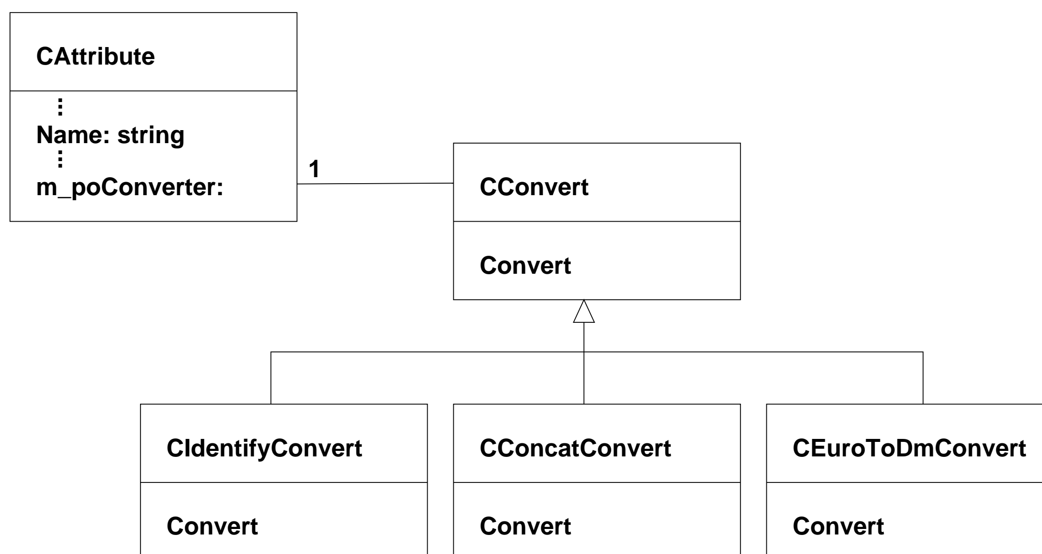

Abbildung 2.3: Die Konvertierungsklassen.

Kapitel 3

Die Spezifikationssprache

3.1 Motivation

Wie bereits in Abschnitt 1.3 angesprochen wurde, werden zu föderierende Datenbankschemata und deren Abhängigkeiten unter Verwendung eines Meta-Datenmodells in *C++* definiert. Die daraus resultierende Struktur ist der Föderierungsgraph.

Die Knoten des Graphen sind Schemata und deren Elemente. Kanten werden durch jeweils zwei miteinander verbundene Schemaelemente gebildet, wobei zu beachten ist, daß der Graph mehrschichtig ist. Schemata besitzen Typen (Klassen, Listen, Zeiger), Klassen besitzen Attribute. Abhängigkeiten müssen jeweils auf Schema-, Typ- und Attributebene definiert werden, wie in Abbildung 3.1 skizziert ist.

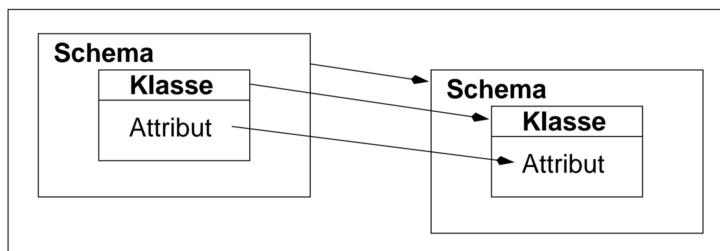

Abbildung 3.1: Abhängigkeiten auf Schema-, Typ- und Attributebene.

Für den Entwurfsprozeß föderierter Datenbanken hat der direkte Einsatz von *C++* zur Spezifikation von Föderierungsgraphen einige Nachteile:

Übersichtlichkeit: Die mittels des Meta-Datenmodells angelegten Schemata und deren Abhängigkeiten sind nicht zuletzt aufgrund geschachtelter Methodenaufrufe schwer lesbar, was zum nächsten Punkt führt.

Verständlichkeit: Um den Aufbau des Föderierungsgraphen nachvollziehen zu können, sind sowohl Kenntnisse in *C++* als auch des Meta-Datenmodells unabdingbar.

Fehleranfälligkeit: Im Meta-Datenmodell existiert kein Mechanismus der verhindert, daß Abhängigkeiten zwischen nicht zusammengehörigen Schemata definiert werden, beispielsweise zwischen Schemata derselben Ebene.

Zur Beseitigung dieser Nachteile könnte eine geeignete Sprache eingesetzt werden, mit der sich ein Föderierungsgraph formal spezifizieren läßt, bevor dieser mittels eines entsprechenden Compilers in *C++*-Code übersetzt wird. Zur Graphspezifikation existieren bereits einige Werkzeuge, die hier jedoch nicht eingehender betrachtet werden sollen:

- GraphBase [Knu93]
- Progres [SWZ95]
- DoDL [Dob96]

Der Grund, daß im Folgenden keine der obigen Graphspezifikationsmechanismen zur Konstruktion föderierter Datenbanken herangezogen wird, liegt in dem Wunsch, eine Sprache zur Verfügung zu haben, die alle genannten Nachteile des Meta-Datenmodells verdeckt.

Eine solche Spezifikationssprache soll über eine kompakte Syntax und einfache Semantik verfügen und damit schnell erlernbar, sowie gut lesbar sein. Übersichtlichkeit und leichte Verständlichkeit erlauben einen Einsatz als Diskussionsgrundlage während des Entwicklungsprozesses. In diesem Zusammenhang erscheinen die oben erwähnten Spezifikationsmechanismen für diese Ziele als zu komplex. Zusätzlich sollen über die Erzwingung von Strukturiertheit mittels geeigneter Mechanismen eine leichte Wartbarkeit von Spezifikationen realisiert werden. Ebenso sind Maßnahmen zu treffen, die Fehler bei der Spezifikation von Abhängigkeiten nach Möglichkeit ausschließen.

In den folgenden Abschnitten wird eine Sprache zur Definition föderierter Datenbankschemata und deren Abhängigkeiten beschrieben. In Abschnitt 3.2 werden zunächst Anforderungen an die Sprache konkretisiert. Abschnitt 3.3 befaßt sich mit Entwurfsentscheidungen, bevor in Abschnitt 3.4 die Konstrukte der Spezifikationssprache beschrieben werden. In Abschnitt 3.5 werden weitere Überlegungen zur Spezifikationssprache angestellt.

3.2 Anforderungen an die Spezifikationssprache

Um der Spezifikationssprache eine möglichst große Ausdruckskraft zu verleihen und die Handhabung der Spezifikationssprache möglichst einfach zu gestalten, sollen einige Anforderungen an die Spezifikationssprache aufgestellt werden:

Plattformunabhängigkeit: Die Spezifikationssprache sollte unabhängig von dem System sein, welches die späteren Abhängigkeiten verwaltet und speichert. Dies abstrahiert von Einschränkungen, die es eventuell bei speziellen Programmiersprachen oder Implementierungsmethoden geben könnte. Ebenso sollte die Sprache auch nicht spezifisch für die später verwendete Datenbank sein, so daß eine Plattformunabhängigkeit auch in dieser Hinsicht sinnvoll sein kann.

Lesbarkeit und Verständlichkeit: Meist geben Schemata nur einen kleinen Ausschnitt der Semantik der ganzen Anwendung wieder. Dies ist auch ein Grund dafür, daß ein automatisches Bestimmen von Abhängigkeiten nicht möglich ist. Wenn nun Abhängigkeiten von Hand konstruiert werden, so wird es meist der Fall sein, daß die Erläuterungen dazu sehr mager ausfallen werden und je komplexer die Abhängigkeiten werden, um so unübersichtlicher wird die Definition. Aus diesem Grunde sollte die Spezifikationssprache eine gute Lesbarkeit und Verständlichkeit der Definition unterstützen oder zumindest ermöglichen. Im Rahmen dieses Punktes

scheint es deshalb auch sinnvoll, daß sich bei der Verwendung der Spezifikationssprache z.B. die Dreistufigkeit des Graphen aus Abbildung 2.2 auf Seite 9 wiederspiegelt.

Natürlich ist Lesbarkeit und vor allem auch Verständlichkeit ein sehr subjektives Empfinden und daraus folgt, daß vielleicht nicht jeder, der diese Spezifikationssprache später einmal benutzt, bzw. eine Spezifikation zu lesen bekommt, diese auch als leicht verständlich erachtet.

Create, Update und Delete: Alle drei Datenbankoperationen sollten unterstützt werden, damit in einer späteren Anwendung keine Einschränkungen bezüglich der Funktionalität entstehen. Dies bedeutet an dieser Stelle nur, daß die Algorithmen, die später auf dem erzeugten Graphen arbeiten, in ihrer Funktionalität nicht eingeschränkt sein sollten.

Zusammensetzbarkeit der Objekte (nesting): In Fällen, in denen Objekte im föderierten Schema aus verschiedenen Objekten eines Komponentenschemas zusammengesetzt sind, sollte auch dies von der Spezifikationssprache unterstützt werden. Als Beispiel sei der Fall angeführt, daß im Komponentenschema nur das Attribut `Name` besteht und als Inhalt den Namen und den Vornamen enthält, im Exportschema allerdings zwei getrennte Attribute `Name` und `Vorname` bestehen. Die Verarbeitung dieses Problems sollte die Sprache ermöglichen.

Wertekonvertierungen von Objekten: Dieser Fall hängt eng mit dem vorhergenannten Punkt zusammen. Falls es nötig sein sollte, Inhalte von Attributen bei der Föderierung zu konvertieren, so sollte die Sprache dieses Vorgehen ermöglichen bzw. definieren. Gemeint sind hiermit z.B. Konvertierungen von DM nach EURO oder Gallonen in Liter.

Unter anderen dienten diese Punkte auch in [SK96] als Entwicklungsgrundlage für eine Sprache und als Ergebniskontrolle.

3.3 Entwurfsentscheidungen

Beim Entwurf der Spezifikationssprache wurde zunächst überlegt, welche Strukturen die Sprache wiedergeben können muß und auf welche Art und Weise dies dann geschehen soll. Dabei können unterschiedliche Strategien verfolgt werden, die zu unterschiedlichen Sprachkonstrukten führen. In diesem Abschnitt werden diese Überlegungsansätze erläutert und gegenübergestellt. Dabei werden nach Möglichkeit auch bereits die Anforderungen aus Abschnitt 3.2 eingebracht. Diese Überlegungen führen schließlich zu dem letztendlich verwendeten Entwurf für die Spezifikationssprache.

In [SK96] wurde eine Sprache vorgestellt, die ebenfalls für die Verbindung mehrerer heterogener Datenbankschemata gedacht ist. Dort sind bidirektionale Verbindungen allerdings der Regelfall. Aus der Abbildung 2.2 auf Seite 9 wird eher das Gegenteil deutlich. Die Abbildungen der hier verwendeten Datenstruktur werden in der Regel in beide Richtungen getrennt definiert. Dies wird auch durch die Verwendung von *Import-* und *Exportschema* verdeutlicht. Die Sprache sollte demnach derart ausgelegt sein, daß sie die in Abbildung 2.2 (Seite 9) gezeigten *Wege* innerhalb der Datenstruktur widerspiegelt. Des weiteren umfaßt [SK96] noch viel mehr Funktionalität (z.B. Spezifikation von Datenbankzugriffen), die auch fest in der Syntax verankert sind und deshalb die Sprache für diesen Anwendungsfall unbrauchbar macht.

Eine der Hauptaugenmerke beim Entwurf dieser Sprache soll die leichte, intuitive Verständlichkeit und Erlernbarkeit sein. Sie soll vor allem auch als leicht auffaßbare Diskussionsgrundlage während des Entwurfprozeßes eines FDBS dienen. Von daher ist es wichtig, daß vor allem die Spezifikationen

der Abhängigkeiten zwischen den Schemata klar sichtbar sind und nicht durch unnötige Angaben überladen oder verdeckt werden.

Es gibt in der Regel den *objektorientierten* und den *klassischen* Ansatz, um beliebige Graphstrukturen zu beschreiben. Als *klassischer* Ansatz ist hierbei eine Art prozedurale Beschreibung gemeint, in der jedes vorkommende Element der Struktur einfach beschrieben wird. Doppelte oder ähnliche Elemente werden hierbei nicht zuvor in Klassen zusammengefaßt und es wird deshalb auch nicht von Vererbung Gebrauch gemacht. Die dabei verwendeten Sprachmittel sind jedoch sehr stark darauf ausgelegt die Struktur an sich zu spezifizieren und nicht die Semantik der Datenstruktur widerzugeben. Spezifikationen nach diesen Vorgehensweisen führen demnach sehr schnell zur Unlesbarkeit. Sie verbinden die Defintionen der einzelnen Objekte mit deren Verbindungen zu anderen Objekten. Später entwickelte Sprachen [SWZ95] erweitern diese Vorgehensweisen noch um die Angabe von Zugriffsmethoden auf einzelne Komponenten, so daß die Sprache noch undurchschaubarer wird. Derartige Sprachen eignen sich für den Entwurf sehr komplexer Datenstrukturen, die sehr stark mit Funktionen behaftet sind.

In dem hier vorliegenden Fall jedoch steht die Art der Datenstruktur bereits fest. Es ist ein recht überschaubarer statischer Graph, der sich hauptsächlich dadurch auszeichnet, daß viele Objekte des Graphen in bestimmter Weise mittels nichtattributierter Kanten miteinander verbunden sind. Eine Spezifikation eines solchen Graphen ist also leicht dadurch zu beschreiben, daß zunächst die im Graphen verwendeten Objekte beschrieben werden und anschließend die Abhängigkeiten zwischen diesen Objekten beschrieben werden können. Diese Vorgehensweise hat drei bedeutende Vorteile:

- Sie entspricht genau der Vorgehensweise, die man bei der Generierung eines solchen Graphen verfolgt.
- Sie ist auf Grund der Zweiteilung leicht zu verstehen und vor allem der Teil der Abhängigkeiten kann sehr gut als Diskussionsgrundlage bei dem Entwurf eines FDBS verwendet werden.
- Es ergibt sich eine Sprache mit kleiner, kompakter Syntax und Semantik.

Für diesen Anwendungsfall ist das von großer Bedeutung. Es geht bei der Spezifikation der Graphstruktur nicht jedesmal um eine Neuschaffung einer Graphstruktur, sondern lediglich um die Anpassung einer vorhandenen Struktur an einen speziellen Anwendungsfall. Dies bedeutet, daß vieles an Semantik bei der Spezifikation schon feststeht und nicht erst beschrieben werden muß. Demnach kann auch besser eine eigens dafür entwickelte Sprache Verwendung finden.

Diese Spezifikationssprache teilt sich also grob in zwei Teile. Sie definiert zunächst die in dem Graphen verwendeten Objekte und anschließend werden diese Objekte miteinander verbunden.

Bei der Definition der Objekte können ebenfalls unterschiedliche Strategien verfolgt werden. Der Graph besteht aus verschiedenen Schemata. Innerhalb eines solchen Schemas werden verschiedene Typen spezifiziert, u.a. auch Klassen. Diese enthalten letztendlich eine Menge von Attributen. Diese Ineinanderschachtelung von verschiedenen Objekten deutet eigentlich sehr stark darauf hin, daß für die Spezifikation solcher Objekte auch ein objektorientierter Ansatz Verwendung finden sollte. Davon wird allerdings Abstand genommen. Eine solche Vorgehensweise würde in bestimmten Fällen allerdings unweigerlich zur Bildung von Superklassen und Vererbung führen. Dies würde sicherlich den Schreibaufwand der Spezifikation in bestimmten Fällen verringern, allerdings steht dem gegenüber die leichte Lesbarkeit der Spezifikation. Außerdem eignen sich derartig verfaßte Spezifikationen nur recht mühsam als Diskussionsgrundlage. Zuviele Referenzierungen würden diesen Vorgang hemmen.

Sogenannte Views könnten den Schreibaufwand bei der Spezifikation ebenfalls mindern und ein Einsatz wäre in dieser Hinsicht auch sinnvoll. Allerdings hat sich auch hier die erhöhte Schwierigkeit beim Lesen der Spezifikation als Nachteil erwiesen. Die Spezifikation ist nicht mehr intuitiv lesbar. Der Einarbeitungsaufwand in die Sprache würde sich dadurch auch nur unnötig erhöhen.

Dies läßt den Schluß zu, daß die an dem Graphen beteiligten Klassen nach der *klassischen* Methode besser spezifiziert werden können. Die genaue Realisierung der sprachlichen Mittel für diesen Teil wird in Abschnitt 3.4 besprochen.

Im zweiten Abschnitt der Spezifikation müssen nun die so definierten Objekte miteinander verbunden werden. Abbildung 2.2 (Seite 9) zeigt dies exemplarisch für die beteiligten Schemata. Um während des Entwicklungsprozesses eines FDBS eine bessere Verbindung zwischen derartigen Grafiken und der Spezifikation zu erlangen, sollte die Sprache derart ausgelegt sein, daß die in der Abbildung 2.2 gezeigten *Wege* von der Spezifikation wiedergegeben werden.

Aus den bisherigen Überlegungen erscheint die Definition in einzelnen Blöcken als sehr sinnvoll. Abbildung 3.2 zeigt eine solche Möglichkeit für den aus Abbildung 2.2 bekannten Graphen auf.

```
FS importiert von ES1 und ES2
IS1 importiert von FS
CS1 importiert von IS1
ES1 importiert von CS1
IS2 importiert von FS
CS2 importiert von IS2
ES2 importiert von CS2
```

Abbildung 3.2: Beispiel für Definition der Abbildungen in einzelnen Blöcken.

Es ist zu erkennen, daß hierbei nur Importe definiert werden, also nur der Fluß entlang der Pfeile aus Abbildung 2.2 wiedergegeben wird. Dies ist nicht besonders übersichtlich und gibt auch nicht die Unterteilung in Import und Export wieder. Als eine Alternative wäre somit eine Top-Down-Sicht zu diskutieren. Bei dieser würde dann nicht entlang der Pfeile definiert, sondern bei den Exportschemata in entgegengesetzter Richtung. Abbildung 3.3 zeigt eine solche Definition, die sicherlich eine bessere Lesbarkeit und ein besseres Verständnis in Hinsicht auf den Graphen mit sich bringt. Als Grundlage dient auch hier Abbildung 2.2.

```
FS importiert von ES1 und ES2
ES1 importiert von CS1
FS exportiert nach IS1 und IS2
IS1 exportiert nach CS1
ES2 importiert von CS2
IS2 exportiert nach CS2
```

Abbildung 3.3: Beispiel für eine Top-Down-Definition in einzelnen Blöcken.

Durch diese Vorgehensweise entfällt eine Defintion gegenüber Abbildung 3.2. Dies kommt dadurch zustande, daß das föderierte Schema nun nicht nur aus zwei Schemata importiert, sondern auch in zwei Schemata exportiert.

Um nun der Dreistufigkeit des Graphen (siehe Abbildung 2.2) zu genügen, ist eine weitergehende Entwurfsidee sinnvoll. Bislang wurden die Definitionen immer zweistufig gehalten, d.h. es wurden

immer nur zwei Schemata miteinander verbunden. Somit könnte man einen Definitionsblock auch als eine solche Hintereinanderschaltung von zwei Abbildungen ansehen. Damit ergibt sich eine Definition wie sie in Abbildung 3.4 gezeigt wird.

```
FS importiert von CS1 via ES1
FS importiert von CS2 via ES2
FS exportiert nach CS1 via IS1
FS exportiert nach CS2 via IS2
```

Abbildung 3.4: Beispiel für eine 3-stufige Top-Down-Definition.

Die Top-Down-Überlegung von vorher sei auch hier angewendet. Diese Art der Darstellung hätte den Vorteil, daß sie der Übersicht sehr zu Gute käme und die vier *Stränge* des Graphen aus Abbildung 2.2 (Seite 9) ebenfalls widerspiegelt. Ein weiterer Vorteil dieser Methode ergibt sich aus der Tatsache, daß die Namensgebung der beteiligten Schemata nicht so semantisch eindeutig gewählt sein könnte wie in diesem Beispiel. Allerdings würde sich ein *Strang* des Graphen mittels Abbildung 3.4 immer deutlich herausprägen. In den Abbildungen 3.2 und 3.3 wäre dieser Zusammenhang dann schon nicht mehr so leicht erkennbar. Außerdem werden die Schemanamen nicht so häufig wiederholt, was Fehlerquellen begrenzt. Auch wird der Entwickler des Graphen durch diese Vorgehensweise geführt, denn er muß immer einen kompletten Strang des Graphen definieren und kommt somit nicht so schnell in die Situation Definitionen zu vergessen.

Die Mehrheit der Vorteile dieser Definitionsweise haben schließlich dazu geführt, daß sie für die weiteren Entwurfsentscheidungen als Grundlage dient.

Aus Abschnitt 2.4 ist bekannt, daß in der verwendeten Datenstruktur nicht nur Schemata untereinander verbunden werden, sondern auch Typen und Attribute. Diese Verbindungen verlaufen nach demselben Prinzip wie die der Schemata, die bislang betrachtet wurden. Auch hier werden jeweils drei Komponenten untereinander verbunden. Vor allem werden nur zwischen solchen Typen und Attributen Verbindungen definiert, bei denen auch deren zugehörige Schemata jeweils untereinander verbunden sind. Abbildung 3.5 verdeutlicht diesen Sachverhalt exemplarisch anhand einer Abbildung von einem Komponenten- über ein Export- hin zu einem föderierten Schema.

Auf Grund der schematischen Ähnlichkeit der Abbildungen können somit auch für die Abbildungen von Typen und Attributen dieselben Sprachkonstrukte als Vorlage dienen. Abbildung 3.5 zeigt darüber hinaus auch die Verschachtelung von Schemata, Typen und Attributen. Verbindungen untereinander sind jeweils nur dann sinnvoll, wenn auch die umgebende Struktur untereinander verbunden ist. Aus diesem Sachverhalt geht hervor, daß auch in der Spezifikationssprache eine solche Schachtelung vorliegen sollte.

Da nun verschiedene Im- und Exporte unterschieden werden müssen, werden für die Spezifikationssprache einige Befehlswörter eingeführt, die im Folgenden groß geschrieben werden. Ihre Semantik dürfte aus den zuvor geführten Erläuterungen bereits jetzt eindeutig sein. Ihre genaue Beschreibung folgt aber erst in Abschnitt 3.4 und dient hier nur zur Veranschaulichung. Bei den Abbildungen wird nun also in die Abbildung von Schemata, von Typen und von Attributen unterschieden. In Abbildung 3.6 werden die Verbindungen aus dem Beispiel der Abbildung 3.5 dargestellt und es wird zum ersten Mal ein genaueres Bild der Spezifikationssprache ersichtlich.

Durch diese Sprachvorgabe können nun alle *normalen* Abhängigkeiten spezifiziert werden, d.h. alle Abhängigkeiten, bei denen keine Konvertierungen der späteren Inhalte und kein Nesting von Nöten

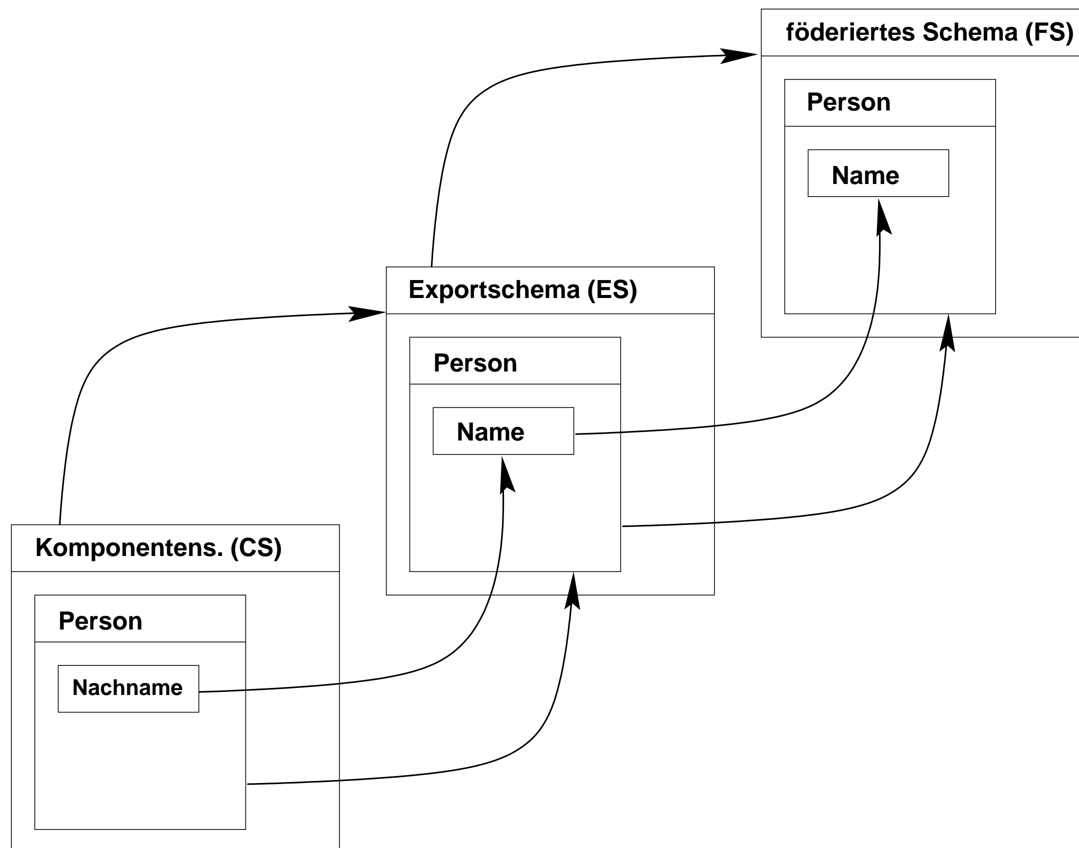

Abbildung 3.5: Schematische Beispielabbildung.

```
SCHEMA_MAPPING
  BRANCH Beispielabbildung
    MAP_SCHEMA FS FROM CS VIA ES
      MAP_TYPE Person FROM Person VIA Person
        MAP Name FROM Nachname VIA Name END_MAP
      END_MAP_TYPE
    END_MAP_SCHEMA
  END_BRANCH
END_SCHEMA_MAPPING
```

Abbildung 3.6: Definition der Verbindungen aus Abbildung 3.5.

ist. Da dies aber auch zu den Anforderungen an das FDBS und somit auch an den Graphen gehört (vgl. Abschnitt 3.2), müssen noch weitere Sprachelemente eingeführt werden.

Bei der Spezifikation des Föderierungsgraphen durch die Projektgruppe 290 [Pro97] wurde festgelegt, daß jegliche Art von Konvertierungen und Umbenennungen nur bei dem Schritt vom Komponentenschema zum Export- bzw. Importschema stattfinden dürfen. Umbenennungen von Attributen können bereits durch die vorhandenen Sprachkonstrukte abgedeckt werden, wie das Beispiel 3.6 zeigt. Für die spätere Konvertierung von Attributinhalten wird bei den Attributabbildungen ein Befehlswort `PROC` eingeführt. Dieses spezifiziert dabei die Methode, die später benutzt werden soll, um die Konvertierung durchzuführen. Auch dieses Sprachmittel wird in Abschnitt 3.4 näher beschrieben und dient an dieser Stelle nur der Veranschaulichung. Es sei in diesem Zusammenhang auch darauf hingewiesen, daß mittels dieses Befehlswortes das geforderte Nesting (vgl. Abschnitt 3.2) ebenfalls bewerkstelligt werden kann und somit keine weiteren sprachlichen Mittel mehr notwendig sind.

Abbildung 3.7 zeigt exemplarisch die Verwendung dieses Befehlswortes.

```
...
MAP Menge
  FROM Menge
  VIA Menge
  PROC GallonenToLiter
END_MAP
...
```

Abbildung 3.7: Verwendung des Befehlswortes 'PROC'.

Zusammenfassend läßt sich also feststellen, daß sich mittels dieser wenigen Sprachmittel eine sehr kleine und kompakte Sprache aufbauen läßt, die den Anforderungen des leichten Verständnisses und der guten Lesbarkeit genauso gerecht wird wie die der Funktionalität im Bezug auf Konvertierungen und Nesting.

3.4 Beschreibung der Sprache

In diesem Abschnitt wird die Syntax und Semantik der Spezifikationssprache beschrieben und anhand einiger Beispiele veranschaulicht. Die zugehörige Grammatik in *Backus-Naur Form* (kurz: BNF), wie auch eine Auflistung aller Schlüsselwörter, ist in Angang A auf Seite 86 zu finden. Zunächst seien

an dieser Stelle einige Konventionen aufgeführt, die beim Entwurf einer Spezifikation mit Hilfe der Sprache zu beachten sind:

- Alle Schlüsselwörter werden grundsätzlich groß geschrieben.
- Anweisungen werden **nicht** mit einem Semikolon abgeschlossen.
- Kommentare werden mit einem Doppelkreuz '#' eingeleitet und reichen von dort bis zum Zeilenende.

Die Graphspezifikation ist in zwei Bereiche aufgeteilt, der Schemadefinition und der Definition von Abhängigkeiten.

3.4.1 Schemadefinition

Der Schemadefinitionsblock wird mit SCHEMA_DEFINITION eingeleitet und mittels END_SCHEMA_DEFINITION beendet. In einem solchen Block können beliebig viele Schemata definiert werden. Anhand eines Beispiels soll die Definition eines Schemas beschrieben werden.

```
1  SCHEMA CS.Beispiel
2    CLASS test
3      String name
4    END_CLASS
5    POINTER ptest TO test
6    LIST tests OF ptest
7  END_SCHEMA
```

1. SCHEMA leitet die Definition eines Schemas ein. Nach dem Schlüsselwort SCHEMA folgt der Schematyp (CS, ES, IS, IES, FS) und ein Schemaname, getrennt durch einem Punkt.
2. CLASS leitet die Definition einer Klasse ein.
3. Durch Angabe eines Typs und eines Namens wird ein Attribut definiert. Der Typ muß definiert sein und kann demnach einer der standardmäßig definierten Typen, oder ein mittels CLASS, POINTER oder LIST selbstdefinierter Typ sein.
4. END_CLASS schließt eine Klassendefinition ab.
5. Zeiger werden mittels POINTER <name> TO <typ> definiert. <name> kann frei gewählt werden, <typ> muß ein Standard- oder selbstdefinierter Typ sein.
6. Die Definition von Listen mittels LIST <name> OF <typ> erfolgt analog zur Zeigerdefinition.
7. END_SCHEMA_DEFINITION beendet die Definition eines Schemas.

3.4.2 Definition von Abhängigkeiten

Nach der Definition von Abhängigkeiten können diese miteinander verbunden werden. Über SCHEMA_MAPPING wird der Abschnitt zur Abhängigkeitsdefinition begonnen und mittels END_SCHEMA_MAPPING beendet.

Abhängigkeiten werden immer in mehreren Stufen — auf Schema-, Typ- und Attributebene — gesetzt, wobei die Richtungen $CS \to FS$ und $FS \to CS$ unterschieden werden.

Hier soll wieder mit Hilfe eines Beispiels das Vorgehen erläutert werden.

```
1  BRANCH Beschreibung
2    MAP_SCHEMA FS.Beispiel FROM CS.Beispiel VIA ES.Beispiel
3      MAP_TYPE test FROM test VIA test
4        MAP name FROM name VIA name END_MAP
5      END_MAP_TYPE
6    END_MAP_SCHEMA
7  END_BRANCH
```

1. BRANCH leitet einen Block zur Definition von Abhängigkeiten zwischen Komponenten-, Export- und föderierten Schemata ein.

2. Abhängigkeiten werden gleichzeitig zwischen drei Schemata für einen Pfad $CS \to FS$ oder $FS \to CS$ definiert. Das erste Schema ist immer ein föderiertes Schema, gefolgt von einem Komponentenschema. Anschließend muß ein Export- oder Import/Exportschema angegeben werden, das im Pfad $CS \to FS$ liegen soll. Über das Schlüsselwort FROM wird bestimmt, daß Abhängigkeiten für die Richtung $CS \to FS$ definiert werden sollen. Für die umgekehrte Richtung $FS \to CS$ ist stattdessen TO einzusetzen. Zusätzlich sind nach VIA entsprechend nur ein Import- oder Import/Exportschema erlaubt.

3. MAP_TYPE folgt jeweils auf ein MAP_SCHEMA und definiert Abhängigkeiten auf Typebene. Die Reihenfolge der Typen entspricht derjenigen der Schemata, die zuvor über MAP_SCHEMA festgelegt wurde, also zuerst ein Typ aus dem föderierten Schema, dann ein Typ aus dem Komponentenschema und zuletzt ein Typ aus dem Schema der Exportebene.

4. Das einfache Verbinden von Attributen ohne Nesting oder Wertekonvertierung erfolgt analog zur Abhängigkeitsdefinition auf Typebene. END_MAP schließt dabei das Attributmapping ab. Die Richtung $CS \to FS$ oder $FS \to CS$ wird auf Schemaebene durch MAP_SCHEMA festgelegt und muß für alle Ebenen gleich sein, d.h. ein MAP_TYPE x FROM y VIA z kann nicht gefolgt werden durch ein MAP x TO y VIA z.

5. Durch END_BRANCH wird ein Block zur Abhängigkeitsdefinition zwischen Schemata beendet.

3.4.3 Nesting/Unnesting

Für das Nesting von Attributen sind die Pfade $CS \to FS$ und $FS \to CS$ genauer zu unterscheiden, wie aus Abbildung 3.8 ersichtlich. Bei einem Pfad $CS \to FS$ ist demzufolge nach dem Schlüsselwort FROM die Menge der Attribute aus einer Klasse des Komponentenschemas anzugeben, die auf ein Attribut im Schema der Exportebene abgebildet werden sollen. Für den Pfad $FS \to CS$ ist umgekehrt die Menge der Attribute aus föderiertem Schema und dem Schema der Exportebene anzugeben,

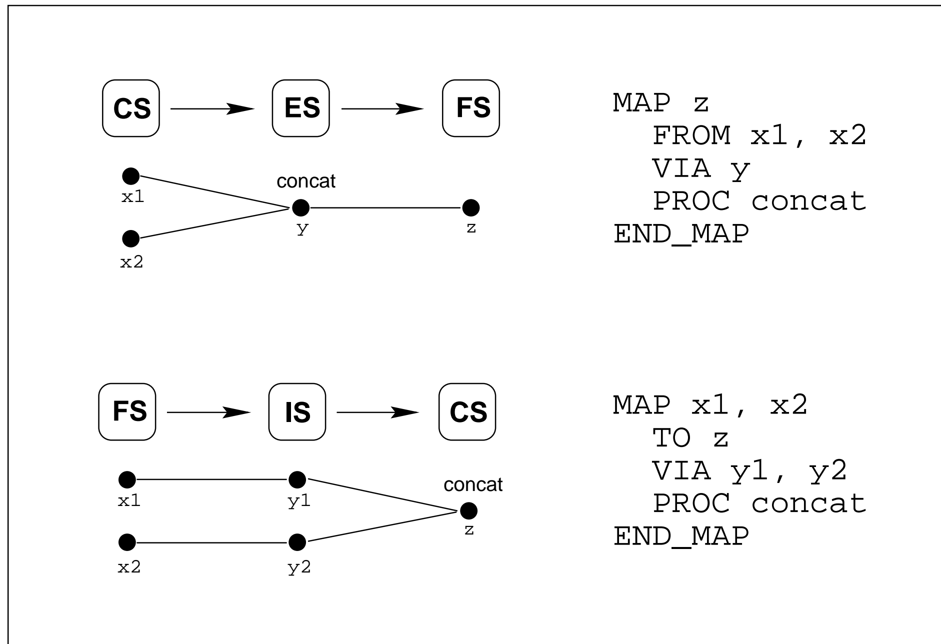

Abbildung 3.8: Nesting für die Pfade $FS \to CS$ and $CS \to FS$.

die in ein Attribut des Komponentenschemas abgebildet werden sollen. Der Name einer Konvertierungsfunktion ist über `PROC <funktionsname>` nach `VIA` zu definieren.

Unnesting entspricht semantisch der Wertekonvertierung, wobei statt einer Konvertierungsfunktion eine Funktion zur Ermittlung eines Teils des entsprechenden zusammengesetzten Attributs anzugeben ist.

3.4.4 Wertekonvertierung

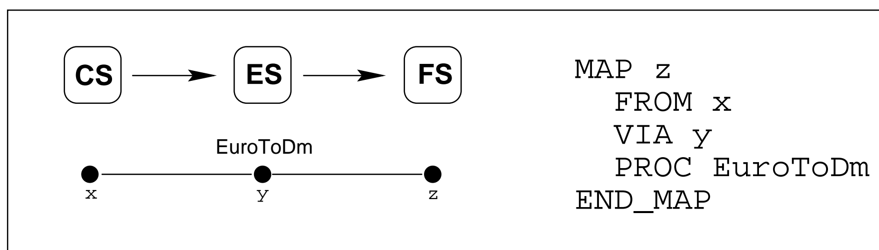

Abbildung 3.9: Konvertierung von Euro nach DM zwischen *CS* und *ES*.

Zur Konvertierung von Attributwerten genügt es, analog zum Nesting mittels `PROC <funktionsname>` den Namen einer Konvertierungsfunktion zu setzen. Der Typ der Funktion spielt dabei in der Spezifikationssprache keine Rolle, da die Funktion selbst in der Datei *graph.H* in *C++* deklariert werden muß. Über den Funktionsnamen wird hier also nur eine entsprechende Funktion aus *graph.H* ausgewählt, wobei auf Groß/Kleinschreibung zu achten ist. Zur Realisierung von Konvertierungsfunktionen sei auf Abschnitt G.2 auf Seite 129 verwiesen. Die Bedeutung der

Datei `graph.H` wird auch in Kapitel 15 auf Seite 70 beschrieben. Abbildung 3.9 zeigt beispielhaft die Angabe einer Funktion zur Konvertierung von Euro nach DM.

3.5 Ausblick

Es gibt noch weitere Überlegungen zur Funktionalität des späteren FDBS. Bei den hier folgenden Punkten steht hauptsächlich der spätere Umgang mit dem FDBS im Vordergrund. Es können verschiedene Situationen auftreten, in denen die Benutzung des System unkomfortabel wird bzw. durch unnötige Replizierung Speicherplatz in den Datenbanken verschwendet wird.

Garbage Collection: Häufig kann es sinnvoll sein, daß Daten zwar zu föderieren sind, diese aber u.U. auf dem entfernten System trotzdem nicht gebraucht werden (Ausnahmefall, Unregelmäßigkeit). Somit würden u.U. in einer Datenbank viele Objekte erzeugt, die überflüssig sind. Eine solche Funktionalität könnte z.B. durch Zeitstempel auf den eingefügten Daten erzielt werden. Für diesen Fall würde es sich dann um eine rein datenbankinterne Lösung des Problems handeln.

Replizieren auf Anfrage: Im Zusammenhang mit der Garbage Collection ist es eventuell auch sinnvoll, daß eine Replikation der Daten nur auf ausdrücklichen Wunsch stattfindet oder ein explizites Unterdrücken der Replikation veranlaßt werden kann. Auch hier wird eine Umsetzung dieser Funktionalität sicherlich in den Aufgabenbereich der entsprechenden Anwendung an der lokalen Datenbank fallen und hat somit keinen direkten Zusammenhang mit dem Föderierungssystem.

Merging, Initialisierung: Das erstmalige Anbinden von bereits bestehenden Datenbanken an das Föderierungssystem wirft ebenfalls Schwierigkeiten auf. Wenn eine Datenbank neu an ein Föderierungssystem angeschlossen wird, so kann diese bereits Daten mitbringen, die durchaus schon equivalent zu bestehenden Daten in anderen angeschlossenen Datenbanken sind. Bei einem ausreichenden Vorwissen über diese Daten könnte u.U. ein automatisiertes Verknüpfen der entsprechenden Daten erfolgen. Hierbei muß sicherlich eine alleinstehende Applikation entwickelt werden, welche die vorhandenen Daten überprüft und dann direkt auf den Föderierungsgraphen zugreifen muß, um die fehlenden Informationen in diesem verfügbar zu machen. Mit der Funktionalität des Graphen an sich hat dies allerdings auch nichts gemein, sondern diese Applikation wird lediglich die bestehende Struktur benuzten.

Späteres Merging: Eine weitere Überlegung ist, auch ein späteres Merging zu erlauben. D.h. zu einem späteren Zeitpunkt definieren zu können, daß bestimmte Objekte in den angeschlossenen Datenbanken dieselben realen Objekte darstellen. Dies wird natürlich nicht von den Datenbanken oder deren Applikationen unterstützt und wäre vermutlich eine Arbeit, die manuell ausgeführt werden müßte, bzw. auch durch eine eigenständige Applikation geregelt werden könnte. Entsprechend gelten dabei die Anmerkungen für das *normale* Merging aus dem vorhergehenden Punkt.

Bei der genaueren Betrachtung dieser Überlegungen stellt sich demnach heraus, daß diese vermutlich zum größten Teil durch zusätzliche Applikationen realisiert werden müssen und die eigentliche Struktur des Föderierungsgraphen keine zusätzliche Unterstützung dabei bieten kann. Die Graphstruktur und somit auch die Spezifikationssprache sind demnach von diesen Überlegungen nicht betroffen. Aus diesem Grund werden die Überlegungen im Rahmen dieser Diplomarbeit nicht weiter diskutiert.

Teil II

Der Compiler

Kapitel 4

Einleitung

Anwendungsprogrammgeneratoren sind Softwaresysteme, die zu den verschiedensten Arten von Problemen Programme zu deren Lösung generieren können. Typische Beispiele für solche Anwendungen sind sogenannte Report-Programme, die aus bestimmten Daten unterschiedlichster Herkunft Protokolle oder Statistiken erzeugen können. Solche Programme können in der Regel systematisch aufgebaut werden, indem die Quelldaten und die Protokolldaten genau beschrieben werden. Um verschiedenste Protokolle oder Statistiken erzeugen zu können, würden die unterschiedlichsten solcher Report-Programme benötigt werden. Ein Anwendungsprogrammgenerator kann solche Programme erzeugen, indem spezifiziert wird, wie die Quelldaten extrahiert werden können und wie das Layout der Ausgabe aussehen soll. Der Begriff des Anwendungsprogrammgenerators wurde ursprünglich für die Definition von Datenbankwerkzeugen in [HKN85] geprägt. Cleaveland [Cle88] generalisierte das Prinzip der Anwendungsprogrammgeneratoren und deutet dabei auf die Zusammenhänge zu Compilern hin. Er beschreibt auch Werkzeuge, die diese Compiler konstruieren. Bei [Kru92] werden Anwendungsprogrammgeneratoren als eine von vielen Möglichkeiten der Software-Wiederverwertung diskutiert. Die Grundlagen und Ideen für die Spezifizierung solcher Anwendungsprogrammgeneratoren wurden schon zu viel früherer Zeit gelegt. Bereits 1952 wurden algorithmische Umwandlungen von Ausdrücken in maschinen-orientierte Formen beschrieben [Rut52], kontextfreie Grammatiken tauchten 1956 [Cho56] auf und wurden zur Spezifikation im Algol 60 Report [alg63] benutzt. Kontextfreie Grammatiken wurden zunächst mittels rekursiven Abstiegs von Hand in Code übersetzt, eine Methode, die auch heute noch angewendet wird [McC72].

Die Eingabe für einen solchen Anwendungsprogrammgenerator kann als eine Spezifikation innerhalb eines genau eingegrenzten Problemfeldes angesehen werden. Sie wird mittels einer vorgegebenen Spezifikationssprache beschrieben. Ein Anwendungsprogrammgenerator ist demzufolge ein Übersetzer für eine solche Sprache. Somit müssen für den Bau von Anwendungsprogrammgeneratoren dieselben Techniken angewandt werden, die für die Spezifikation von Sprachen genutzt werden.

Eli [GHL+92, WHK88, Wai94, Kas94, Kas96] ist ein integriertes Werkzeugset für die Sprachenimplementierung. Es besteht aus einer Vielzahl von generierten Werkzeugen und Bibliotheken, die bei der Wiederverwendung eine große Zahl von Aufgaben aus der Sprachenimplementierung bereits abdecken.

Kapitel 5

Die Aufgabe des Compilers

Der hier spezifizierte Compiler akzeptiert ein Programm, welches in der Sprache aus Kapitel 3 geschrieben ist. Diese Sprache spezifiziert eine spezielle Datenstruktur. Die Aufgabe des Compilers ist es jenes Programm in einen *C++* Sourcecode zu übersetzen, der zu einem späteren Zeitpunkt in der Lage sein wird, eine für den praktischen Einsatz nutzbare Datenstruktur in der objektorientierten Datenbank O_2 anzulegen. Wenn das Programm syntaktische Fehler aufweisen sollte, so soll der Compiler unter Angabe der Zeilen- und Spaltennummer eine entsprechende Fehlermeldung ausgeben.

5.1 Die Übersetzung

Abbildung 5.1 zeigt ein Fragment eines Programmes, welches in der Spezifikationssprache programmiert ist. Es definiert eine Klasse und einige Attribute innerhalb dieser Klasse.

```
SCHEMA CS.Angio
  CLASS patient
    STRING Name
    STRING Vorname
  END_CLASS
END_SCHEMA
```

Abbildung 5.1: Beispielfragment aus einer Eingabespezifikation.

Der Compiler wird das Programm, welches dieses Fragment enthält, akzeptieren und ein Programm erzeugen, welches ein equivalentes Programmfragment in der Sprache *C++* enthalten wird. Dieses Zielfragment soll nicht die oben dargestellte Struktur direkt in *C++* darstellen, sondern ist vielmehr ein Sourcecode zur Generierung einer viel komplexeren Datenstruktur in der Datenbank O_2 (siehe Kapitel 2.2). Die entsprechende Ausgabe soll z.B. wie in Abbildung 5.2 dargestellt aussehen.

In ihr wird ein *C++* -Code gezeigt, der aus fünf Befehlen besteht. Die damit verbundene Semantik ist wie folgt: Es wird ein neues Schema mit dem Namen `CSAngio` angelegt, die für die spätere Verwendung notwendigen `SimpleTypes` werden in diesem Schema angelegt, die Klasse `patient` wird angelegt und anschließend werden die beiden Attribute `Name` und `Vorname` in diese Klasse eingetragen.

```
PCSchema CS_Angio = new Cschema (`` CSAngio'', pg);
BuildSimpleTypes (CS_Angio);
PCComplexType CS_Angio_patient =
  new CComplexType (``patient'', CS_Angio);
PCAttribute CS_Angio_patient_Name =
  new Cattribute(``Name'',CS_Angio->GetType
    (``STRING''), CS_Angio_patient);
PCAttribute CS_Angio_patient_Vorname =
  new Cattribute(``Vorname'',CS_Angio->GetType
    (``STRING''), CS_Angio_patient);
```

Abbildung 5.2: Beispielfragment für einen Ausgabetext.

5.2 Fehlermeldungen

Wenn dem Compiler ein fehlerhaftes Programm übermittelt wird, so muß der Compiler mitteilen, daß dieses Programm fehlerbehaftet ist und dem Benutzer die genaue Position des Fehlers mitteilen. Fehler werden spezifiziert durch den Dateinamen, eine Zeilennummer und die entsprechende Buchstabenposition innerhalb dieser Zeile. Auch sollte eine Beschreibung des Fehlers durch den Compiler erfolgen. Diese Fehlermeldungen werden zur Laufzeit des Compilers durch die Ausgabe "`ERROR`" gekennzeichnet. Allerdings wird der Compilerlauf nicht unterbrochen. Die generierte Ausgabe kann aber unter Umständen nicht richtig ausgeführt werden. Neben diesen Fehlermeldungen werden während der Laufzeit eventuell auch Warnungen ausgegeben. Diese beschreiben jedoch keine Fehler, sondern beinhalten lediglich zusätzliche Informationen.

Kapitel 6

Die Eingabesprache

Der hier beschriebene Compiler akzeptiert eine Sprache, die aus Kapitel 3 bekannt ist. Dieses Kapitel beginnt damit die allgemeine Aufgabe der Sprache kurz zu umschreiben. Es folgt eine informelle Beschreibung der benutzten Vokabeln und eine formale Beschreibung der Struktur eines Programms, welches in dieser Sprache geschrieben wird.

6.1 Allgemeine Beschreibung

Die Sprache dient dazu eine spezielle, aus Kapitel 2.2 bekannte, Datenstruktur zu beschreiben. In dieser Datenstruktur gibt es einen Spezifikationsteil, in dem alle vorkommenden Schemata definiert werden müssen. Ein Schema wird mit einem eindeutigen Namen definiert, der sich aus dem Schematyp und dem eigentlichen Schemanamen zusammensetzt:

```
SCHEMA CS.Angio
...
END_SCHEMA
```

Die Schemadefinition ist als Blockstruktur ausgelegt. Innerhalb eines solchen Schemas können Klassen, Pointer und Listen definiert werden. Die Pointer und Listen sind einfache Typdeklarationen, wie man sie aus anderen Programmiersprachen bereits kennt:

```
POINTER Pname TO Person
LIST Cname TO Person
```

Die Klassendefinitionen sind dagegen wieder als ein Blockkonstrukt ausgelegt. Sie können mit mehreren Attributen gefüllt werden:

```
CLASS Person
  STRING Name
...
END_CLASS
```

Die Attribute können hierbei von einem einfachen, vordefinierten Typ sein oder auch vom Typ Pointer, List oder Class. Darüber hinausgehende Definitionen, wie man sie vielleicht aus anderen Sepezifikationssprachen kennt, sind hier nicht vorgesehen.

Im Abbildungsabschnitt der Sprache werden dann die zuvor definierten Objekte miteinander verbunden, um die in Kapitel 2.2 beschriebene Datenstruktur zu beschreiben. Die Vorgehensweise ist dabei so, daß zunächst die Schemata untereinander verbunden werden. Innerhalb dieser Blockumgebung werden dann die Klassen, Pointer und Listen miteinander verbunden. Auch hier sind die Klassen wieder Blöcke, in denen die Attribute mit Attributen anderer Klassen verbunden werden können. Es ergibt sich also anschaulich eine ähnliche Blockstruktur wie im Spezifikationsteil der Sprache. Eine genaue Beschreibung der Sprache findet sich in Kapitel 3.

6.2 Das Vokabular

Das Vokabular einer Sprache setzt sich aus sogenannten *Basissymbolen* (engl. basic-symbols) und *Kommentaren* zusammen [Wai]. Es gibt drei verschiedenen Arten von Basissymbolen: *Identifizierer* (engl. identifier), *Bezeichner* (engl. Denotations) und *Abgrenzer* (engl. Delimiter). Identifizierer sind Namen, die frei gewählt werden können, um z.B. Typen oder Variablen zu benennen. Bezeichner repräsentieren gewisse Werte, die vom Sprachdesigner festgelegt wurden. Abgrenzer werden benutzt, um einzelne Identifizierer und Bezeichner voneinader zu trennen. Dadurch kann einer Sprache eine bestimmte Struktur mitgegeben werden, die auch zur übersichtlicheren Darstellung eines Programms in dieser Sprache genutzt werden kann.

In der hier verwendeten Sprache wird ein Identifier `name` genannt. Er besteht aus einem Buchstaben, dem eine Kette von weiteren Buchstaben oder Zahlen folgen kann. Als Bindeglied ist auch der Unterstrich "_" erlaubt.

`name`-Beispiel:

```
x, g50, CaseInsensitive, CASEINSENSITIVE, oder_auch_so
```

Es werden in dieser Sprache Unterschiede zwischen Groß- und Kleinschreibung gemacht, so daß sich die oben erwähnten Namen alle voneinander unterscheiden.

Bezeichner, im eigentlichen Sinne, werden in dieser Sprache nicht verwendet. Ein Beipiel für Denotations in anderen Sprachen sind zum Beispiel Zahlen. Hier allerdings werden keine Konstanten verwendet und richtige Inhalte von z.B. Attributen gibt es auch nicht. Es werden lediglich Zuordnungen zwischen Namen und Typen beschrieben.

In der Sprache werden ausschließlich sogenannte *Wortsymbole* als Abgrenzer verwendet. Einige davon wurden im vorhergehenden Kapitel bereits erwähnt, die anderen lassen sich aus Kapitel 3 (Seite 13), bzw. aus der vollständigen Grammatik im Anhang A (Seite 86) ableiten und werden in Abbildung 6.1 zusammengefaßt.

Spezialsymbole, so wie sie aus anderen Sprachen bekannt sind, werden hier nicht als Abgrenzer verwendet. Beipiele für solche Spezialsymbole sind: `;` `,` `+` `-` `*` { } , etc. . Überflüssige Leerzeichen werden ignoriert und können somit auch nicht als Abgrenzer verwendet werden.

Ein *Wortsymbol* kann nicht mehr als *Identifizierer*, z.B. `name` benutzt werden, es handelt sich also um sogenannte reservierte Wörter.

Ein Kommentar in dieser Sprache wird durch ein "#" eingeleitet und geht von dort bis zum Ende der Zeile. Beim Beginn einer neuen Zeile muß der Kommentar also erneut mit einem "#" eingeleitet werden.

Wortsymbole:
SCHEMA_DEFINITION, END_SCHEMADEFINITION, SCHEMA, END_SCHEMA, CLASS, END_CLASS, POINTER, LIST, TO, OF, INVERSE, SCHEMA_MAPPING, END_SCHEMA_MAPPING, BRANCH, END_BRANCH, MAP_SCHEMA, END_MAP_SCHEMA, MAP_TYPE, END_MAP_TYPE, FROM, VIA, MAP, END_MAP, PROC

Abbildung 6.1: In der Sprache verwendete Abgrenzer.

Beispiele für Kommentare:

```
# Dies ist ein einleitender Kommentar
CLASS Person # hier wir die Klasse Person definiert
END_CLASS
```

Wortsymbole und *Identifizierer* sind sogenannte unbeschränkte *Basissymbole*. Dies bedeutet, daß diese durch mindestens ein Leerzeichen voneinander getrennt werden müssen, wenn sie im Zusammenhang verwendet werden:

```
#falsch
CLASSPerson
END_CLASS

#richtig
CLASS Person
END_CLASS
```

6.3 Die Grammatik

Ein vollständiges Programm wird *Satz* (engl. sentence) [Wai] einer Sprache genannt, in der es programmiert ist. Die Struktur eines solchen Satzes wird in Form von sogenannten *Phrasen* beschrieben. Die Übersetzung eines solchen Programms basiert nun auf der Struktur solcher Phrasen, wie sie in einer Grammatik für eine Sprache definiert wird. Die Phrasenstruktur muß dabei sowohl den Weg zum Schreiben eines Programms als auch die Bedeutung des Programms widerspiegeln.

In der Regel ist es angebracht eine Grammatik in verschiedene Teile zu zerlegen, die jeweils voneinander unabhängige Phrasen beschreiben und dann jeden dieser Teile getrennt zu betrachten. Symbole, die nicht mehr weiter aufgeteilt werden, die also in der Grammatik auf der linken Seite stehen und auf keiner rechten Seite benutzt werden, werden *Terminale* (engl. terminals) genannt.

Der Compiler wird mittels des Compilerbauwerkzeugs ELI entwickelt. Die Sprachdefinition nach der Syntax von ELI läßt sich unmittelbar aus der BNF der Grammatik aus Anhang A auf Seite 86 ableiten:

```
specification : schema_specification schema_mapping .

schema_specification :  'SCHEMA_DEFINITION' schema_def+
                        'END_SCHEMA_DEFINITION' .
schema_mapping : 'SCHEMA_MAPPING' path_def+ 'END_SCHEMA_MAPPING' .
```

Die erste Grammatikregel besitzt also zwei Phrasen, `schema_specification` und `schema_mapping`. Eine `schema_specification` besteht aus ein oder mehrerer `schema_def`. Die Abgrenzer `SCHEMA_DEFINITION` und `END_SCHEMA_DEFINITION` werden benutzt um diese Phrase im Programmtext zu trennen. Für `schema_mapping` gilt dies analog.

6.3.1 Definitionsblock - Grammatik

Wie bereits in Abschnitt 6.1 aufgeführt, werden hier mehrere Blockstrukturen miteinander verschachtelt. Ein Schema besitzt einen Namen und ein Menge von einem oder mehreren `class_decl`, welches Klassen-, Pointer oder Listendefinitionen sein können:

```
schema_def : 'SCHEMA' schema_specifier class_decl+ 'END_SCHEMA' .
schema_specifier : schema_type_def '.' schema_name_def .

class_decl      :complex_decl / pointer_decl / compound_decl .
```

Diese Grammatik unterscheidet zwischen der Definition eines Namens und der Verwendung desselben, z.B. `schema_name_def` und `schema_name_use`. Diese Unterscheidung reflektiert die Bedeutung des Programms. Die Klassendefnition ist von der Struktur her ähnlich der Schemadefinition. Die Klasse erhält einen Namen und beinhaltet eine Menge von ein oder mehrerer Attributdefinitionen. Zusätzlich können im Anschluß an eine Klasse noch sogenannte *inverse Attribute* definiert werden:

```
complex_decl: complex_type type_name_def attribute+ 'END_CLASS'
                inverse_decl* .
complex_type: 'CLASS' .

inverse_decl: inverse_type type_name_von_use  '.'
                                     attribute_name_von_use
                           type_name_nach_use '.'
                                     attribute_name_nach_use.
inverse_type: 'INVERSE' .
```

Attributdefinitionen sind entweder `simple_decl`, d.h. sie verwenden einen vorgegebenen Standardtyp, oder `generic_decl`, d.h. sie verwenden eine bereits definierte Klasse, einen Pointer oder eine Liste als Typ:

```
attribute    : simple_decl / generic_decl .

simple_decl  : simple_type attribute_name_def .
generic_decl : type_name_use attribute_name_def .
```

Die Pointer- und Listendefinition ist vom Aufbau her einfacher. Es wird jeweils ein Name vergeben und für den Typ ein bereits definierter, beliebiger anderer Typ angegeben:

```
pointer_decl : pointer_type type_name_def 'TO' type_name_use .
pointer_type : 'POINTER' .

compound_decl : compound_type type_name_def 'OF' type_name_use .
compound_type : 'LIST' .
```

6.3.2 Die Mappingblock - Grammatik

In dem Mappingblock wird zunächst ein *Zweig (branch)* definiert. Dieser ist ausschließlich zur besseren Übersicht, also Struktur des Programms gedacht und hat keine direkten Auswirkungen auf die Übersetzung. Er wird aber in jedem Fall verlangt, um die Programmierung zu führen (siehe auch Abschnitt 3.2, Seite 14):

```
path_def:'BRANCH' path_name_def mapping_schema_source 'END_BRANCH'.
```

Da sich die Abbildungen in die Import- und die Exportrichtung unterteilen, muß in der Grammatik an dieser Stelle ebenfalls in diese beiden Richtungen unterschieden werden. Dies ist vor allem notwendig, um in den folgenden geschachtelten Blöcken eine eindeutige Befehlsstruktur vorzugeben, so daß niemals zwei Richtungen miteinander vermischt werden können:

```
mapping_schema_source : mapping_schema_from / mapping_schema_to .

mapping_schema_from : 'MAP_SCHEMA' schema_specifier_fs
                       'FROM' schema_specifier_cs
                       'VIA' schema_specifier_es mapping_type_from+
                      'END_MAP_SCHEMA' .
mapping_schema_to   : 'MAP_SCHEMA' schema_specifier_fs
                       'TO' schema_specifier_cs
                       'VIA' schema_specifier_is mapping_type_to+
                      'END_MAP_SCHEMA' .
```

Die Richtungen werden also durch die Delimiter `FROM` und `TO` definiert. Der Unterschied bei den `schema_specifier` ist ersichtlich: Bei der Richtung *from* wird ein Schema vom Typ *ES* benutzt, bei der anderen Sicht ein Schema vom Typ *IS*. Dies wird auch hier durch die Verwendung unterschiedlicher Namen kenntlich gemacht. Die jeweils richtig darin verschachtelte Blockstruktur ist das Type-Mapping, in der Klassen, Pointer und Listen jeweils untereinander verbunden werden können:

```
mapping_type_from: 'MAP_TYPE' type_name_use_fs
                    'FROM' type_name_use_cs
                    'VIA' type_name_use_es map_from*
                   'END_MAP_TYPE' .
mapping_type_to  : 'MAP_TYPE' type_name_use_fs
                    'TO' type_name_use_cs
                    'VIA' type_name_use_is map_to*
                   'END_MAP_TYPE' .
```

Im Falle eines Pointer- oder Listenmappings wird die Phrase `map_to` oder `map_from` einfach ausgelassen. In diesen werden dann als letzter geschachtelter Block die Attribute miteinander verbunden. Es wird unterschieden, ob einfach nur Attribute direkt miteinander verbunden werden sollen, oder eventuell Nesting oder Umrechnungen (siehe Abschnitt 3.2) mit berücksichtigt werden müssen. Natürlich wird auch hier konsequent die Richtung auseinandergehalten:

```
map_from      : map_from_simple / map_from_nest / map_from_proc .
map_to        : map_to_simple / map_to_nest / map_to_proc .

map_from_simple : 'MAP' map_select_fs
                        'FROM' map_select_cs
                        'VIA'  map_select_es
                  'END_MAP' .
map_from_nest   : 'MAP' map_select_fs
                        'FROM' map_select_multi_cs ','
                               (map_select_multi_cs // ',')
                        'VIA'  map_select_es
                        proc_decl
                  'END_MAP' .

map_from_proc   : 'MAP' map_select_fs
                        'FROM' map_select_cs
                        'VIA'  map_select_es
                        proc_decl
                  'END_MAP' .

map_to_simple : 'MAP' map_select_fs
                      'TO'  map_select_cs
                      'VIA' map_select_is
                'END_MAP' .

map_to_nest   : 'MAP' fs_part
                      'TO' map_select_cs
                      'VIA' is_part
                      proc_decl
                'END_MAP' .
   fs_part: map_select_multi_fs ',' map_select_multi_fs
            (',' map_select_multi_fs)* .
   is_part: map_select_multi_is ',' map_select_multi_is
            (',' map_select_multi_is)* .

map_to_proc : 'MAP' map_select_fs
                    'TO'  map_select_cs
                    'VIA' map_select_is
                    proc_decl
              'END_MAP' .

proc_decl   : 'PROC' proc_name_def  .
```

```
SCHEMA_DEFINITION

SCHEMA CS.book_1
CLASS Buch
STRING Titel
END_CLASS
END_SCHEMA

SCHEMA ES.book_1
CLASS Buch
STRING Titel
END_CLASS
END_SCHEMA

SCHEMA FS.books
CLASS Buch
STRING Titel
END_CLASS
END_SCHEMA

SCHEMA_MAPPING

BRANCH CS_book_1_FS_books
MAP_SCHEMA FS.books FROM CS.book_1 VIA ES.book_1
MAP_TYPE Buch FROM Buch VIA Buch
MAP Titel FROM Titel VIA Titel END_MAP
END_MAP_TYPE
END_MAP_SCHEMA
END_BRANCH

END_SCHEMA_MAPPING
```

Abbildung 6.2: Beispiel einer Definition und einem dazu passenden Mappingabschnitt.

6.4 Beispiel

Eine Spezifikation gemäß der vorgestellten Grammatik könnte demnach wie in Abbildung 6.2 aussehen. Hier werden lediglich drei Schemata vom Komponentenschema zum föderierten Schema definiert. In diesen existiert jeweils nur die eine Klasse `Buch`, die durch das Attribut `Titel` beschrieben ist. Im Mappingabschnitt werden dann die Schemata, Klassen und Attribute miteinander verbunden.

Kapitel 7

Die Zielstruktur

Die Aufgabe des Compilers ist es, aus einem Programm in der vorhergehend definierten Sprache eine Ausgabe in der Programmiersprache *C++* zu generieren. Die genaue Beschreibung der Klassen und Methoden, die dabei verwendet werden, findet sich in Abschnitt 2.4.

Dieses Kapitel beginnt damit die einzelnen Sprachkonstrukte kurz aufzuführen und die entsprechenden Ausgaben in der Zielsprache aufzuführen ohne weiter auf deren Semantik einzugehen.

7.1 Programmkopf

Unabhängig von der Eingabe erzeugt der Compiler zunächst einen Programmkopf, der für alle Eingabemöglichkeiten stets gleich bleibt. Basierend auf *C++* werden hier benötigte Dateien eingebunden und grundlegend benötigte Variablen definiert und deren Existenz überprüft. Die Abbildung 7.1 zeigt diesen Ausschnitt aus dem *C++* -Quellcode, der vom Compiler ausgegeben wird. Es zeigt sich die typische Gestalt eines *C++* -Programms: Zunächst wird die notwendige Datei `build.H` eingebunden. In ihr werden die später implementierten Methoden `BuildGraph` und `BuildSimpleTypes` deklariert. Am Ende der Abbildung sieht man den Beginn der Definition der Funktion `BuildGraph`. Der Programmkopf wird hier der Vollständigkeit wegen aufgeführt. Er stammt aus den Ergebnissen der PG290 [Pro97] und ist zwingender Bestandteil der Ausgabe.

7.2 Der Spezifikationsteil

Der Spezifikationsteil der Sprache besteht aus mehreren ineinander geschachtelten Blockstrukturen, die jede für sich zu spezifischen Ausgaben führt.

7.2.1 Schemadefinition

Die erste Blockstruktur, von der mehrere hintereinander auftauchen dürfen und die zu einer Ausgabe führt, ist die Schemadefinition:

```
SCHEMA CS.Angio
...
END_SCHEMA
```

```
//////////////////////////
//
// Buildgraph.C
//
//////////////////////////

#include ``build.H''

void BuildGraph ( PCGraph pg )
{
// require
// assert (pg != NULL );
```

Abbildung 7.1: Der Programmkopf der Ausgabe.

Dieses Beipiel definiert z.B. ein Schema vom Typ Komponentenschema mit dem Namen “Angio”. Der Name des Schemas setzt sich also aus dem Typ des Schemas und dem eigentlichen Namen zusammen. Als Schematypen sieht die Sprache CS, IS, ES, IES und FS vor, welche nur großgeschrieben akzeptiert werden. Die Ausgabe zu diesem Beispiel muß wie folgt aussehen:

```
PCSchema CS_Angio = new ("CSAngio", pg);
BuildSimpleTypes (CS_Angio);
```

Innerhalb der obigen Schemadefinition dürfen Klassen, Pointer und Listen definiert werden.

7.2.2 Klassendefinition

Das Sprachkonstrukt für die Klassendefinition sieht wie folgt aus:

```
CLASS Patient
...
END_CLASS
```

Eine Klasse mit dem Namen “Patient” soll generiert werden. Die dazu vom Compiler nötigen Ausgaben sehen wie folgt aus:

```
PCComplexType CS_Angio_Patient =
       new CComplexType("Patient",CS_Angio);
```

Der Name der erzeugten Klasse setzt sich also nicht nur aus dem oben definierten Namen zusammen, sondern beinhaltet auch den Schematyp und dessen Namen.

7.2.3 Pointerdefinition

Bei der Pointerdefinition wird wie gewohnt eine Pointervariable deklariert, die auf eine Klasse, eine Liste oder einen weiteren Pointer zeigen darf - nicht aber auf ein Attribut. Die Definition sieht wie folgt aus:

```
POINTER Ppatient TO Patient
```

Der Compiler wird darauf folgendermaßen reagieren:

```
PCPointerType CS_Angio_Ppatient =
  new CPointerType (CS_Angio, CS_Angio_Patient);
```

7.2.4 Listendefinition

Die letzte Definitionsmöglichkeit innerhalb eines Schemas ist die Listendefinition. Auch hier ist das Sprachkonstrukt sehr einfach aufgebaut. Es wird ein Listenname definiert und der Typ der Listenelemente angegeben:

```
LIST Cpatient OF Ppatient
```

Eine entsprechende Ausgabe des Compiler lautet wie folgt:

```
PCCollectionType CS_Angio_Cpatient =
  new CCollectionType (CS_Angio,CS_Angio_Ppatient);
```

Die oben beschriebene Klassendefinition bildet eine weitere Blockumgebung, in welcher nun Attribute definiert werden können. Im direkten Anschluß an eine Klasse können wahlweise sogenannte Invers-Attribute definiert werden.

7.2.5 Attributdefinitionen

Die Attributdefinitionen werden aus dem Typ und dem Attributnamen zusammengesetzt. Als Typen sind in der Sprache STRING, CHAR, INTEGER, LONG, REAL, BOOLEAN, TIME und DATE vorgesehen. Die Typangaben müssen wie angegeben groß geschrieben werden. Ein Beispiel für eine solche Attributdefinition sieht demnach folgendermaßen aus:

```
STRING Name
```

Der Compiler muß darauf mit folgender Ausgabe reagieren:

```
PCAttribute CS_Angio_Patient_Name =
  new Cattribute ("Name",CS_Angio->GetType ("STRING"),
                CS_Angio_Patient);
```

Auch hier kann man ersehen, daß der letztendlich vergebene Variablenname sich aus dem Schematyp, dem Schemanamen und dem Klassennamen zusammensetzt, in dem das Attribut definiert ist.

7.2.6 Inversdeklaration

Im unmittelbaren Anschluß nach einer Klassendefinition, also nach dem “END_CLASS”, sieht die Sprache einen folgenden, optionalen Sprachausdruck vor:

```
INVERS Patient.Name Arzt.Name
```

Hiermit können klassenübergreifend inverse Attribute definiert werden, die für die spätere Datenstruktur und deren Verwendung wichtig sein können. Entsprechend die Ausgabe:

```
CS_Angio_Patient_Name->SetInversAttr(CS_Angio_Arzt_Name);
```

7.3 Der Mappingteil

Im Mappingteil der Eingabesprache werden jeweils Schema, Typen und Attribute mit ihresgleichen aus anderen Schemata verbunden. Die Sprache sieht dabei eine ähnliche Blockstruktur wie im Spezifikationsteil vor. Zunächst werden Schemata miteinander verbunden. Innerhalb dieser Blockung werden Klassen, Pointer und Listen verbunden und innerhalb von Klassen werden Attribute miteinander verbunden. Die Schema- und Klassenverbindungen benutzen ein equivalentes Sprachkonstrukt, währenddessen bei den Attributen bestimmte Besonderheiten auftreten können (siehe Abschnitt 6.3.2, Seite 34). Im Groben wird noch die *Richtung* der Verbindung unterschieden, in ein `map_to` und ein `map_from`, deren Unterschiede in der Ausgabe des Compilers im folgenden auch beschrieben werden.

7.3.1 Schemamapping

Das erste Sprachkonstrukt, welches zu einer relevanten Compilerausgabe führt, ist das Schemamapping:

```
MAP_SCHEMA FS.KrKh FROM CS.Angio VIA ES.Angio
...
END_MAP_SCHEMA
```

In diesem Beipiel werden drei Schemata miteinander verbunden. Die Richtung weist vom Komponentenschema CS über das Exportschema ES zum föderierten Schema FS. Dementsprechend führt dies zu folgender Compilerausgabe:

```
CS_Angio->AddPostSchema ( ES_Angio );
ES_Angio->AddPostSchema ( FS_KrKh );
```

7.3.2 Typemapping

Die Ein- und Ausgaben beim Typemapping verlaufen analog zu denen des Schemamappings:

```
MAP_TYPE PatientFS FROM PatientCS VIA PatientES
```

Die hier gewählten Klassennamen sind nur der besseren Übersicht halber mit den Namensendungen FS, ES und CS versehen. In der Praxis kann es durchaus vorkommen, daß in allen drei Schemata Klassen mit denselben Namen definiert worden sind. Der Compiler wird mit

```
CS_Angio_PatientCS->AddPostType ( ES_Angio_PatientES );
ES_Angio_PatientES->AddPostType (FS_KrKh_PatientFS );
```

reagieren.

7.3.3 Attributmapping

Beim sogenannten Attributmapping können nun verschiedene Fälle auftreten. In allen Fällen müssen, wie zuvor beim Type- und Schemamapping, die Attribute entsprechend verbunden werden, welches im einfachsten Fall zu einem sehr ähnlichen Sprachkonstrukt wie oben führen würde. Für den Fall, daß noch eventuelle Wertekonvertierungen (siehe Abschnitt 3.2) oder Zusammenführungen von Attributen (siehe Abschnitt 3.2) hinzukommen, müssen entsprechend mehr Attribute und eine Umwandlungsmethode angegeben werden:

```
MAP NameFS FROM NameCS, VornameCS VIA NameES PROC concat
```

In diesem Fall werden zunächst wie gewohnt die Attribute miteinander verbunden:

```
CS_Angio_PatientCS_NameCS->AddPostAttribute(ES_Angio_PatientES_NameES);
CS_Angio_PatientCS_VornameCS->AddPostAttribute(ES_Angio_PatientES_NameES);
ES_Angio_PatientES_NameES->AddPostAttribute(FS_KrKh_PatientFS_NameFS);
```

Danach wird im Attribut `NameES` die Konvertermethode eingetragen und es müssen die Attribute eingetragen werden, die der Konvertermethode als Quelle dienen:

```
ES_Angio_PatientES_NameES->set_converter ( new CconcatConvert () );
ES_Angio_PatientES_NameES->AddPostParameter(CS_Angio_PatientCS_VornameCS);
ES_Angio_PatientES_NameES->AddPostParameter(CS_Angio_PatientCS_NameCS);
```

In den Beispielen wurde immer *FROM* als Mappingrichtung angegeben. Für den Fall, daß *TO* verwendet wird, ähneln sich die Compilerausgaben. Anstelle von 'AddPost...' wird 'AddPre...' ausgegeben. Die Angabe von 'AddPostParameter' bleibt gleich, weil es sich einfach nur um die Einrichtung einer Liste handelt, die von der Richtung zunächst nicht unmittelbar betroffen ist.

7.4 Kommentare

Innerhalb der Eingabesprache können Kommentare verwendet werden. Die Kommentare beginnen mit einem "#" und gehen von dort bis zum Ende der Zeile. Der Compiler akzeptiert solche Kommentare und sie führen zu keinerlei Ausgaben, sondern werden einfach ignoriert.

Kapitel 8

Aufbau des Compilers mit Eli

Eli verfügt über eine reichhaltige Sammlung an Werkzeugen, welche für den Compilerbau notwendig sind. Es unterstützt die gesamtheitliche Entwicklung und sollte für die meisten Probleme von Übersetzungen bereits Lösungen mitbringen. Bei Eli wird angenommen, daß die Probleme, die sich bei der Übersetzung stellen, in kleine Teilprobleme zerlegt werden. Diese Teilprobleme werden getrennt betrachtet und gelöst.

Abbildung 8.1 zeigt eine modulare Aufteilung eines typischen Compilers [Wai94]. Jeder der Blöcke könnte von einem oder mehreren Werkzeugen generiert werden. Zum Beispiel, *LEX* [LS] und *YACC* [Joh] (bzw. *FLEX* oder *BISON*) können mittels deklarativer Beschreibungen den Teil der *strukturellen Analyse* generieren. Jeder dieser Blöcke stellt ein Teilproblem bei der Übersetzung dar, wobei es Eli ermöglicht die Teillösungen miteinander zu verbinden und zu einem Gesamtergebnis zusammenzutragen, welches die gewünschte Übersetzung vollbringt.

Der Compiler, der von Eli generiert wird, liest eine Eingabedatei, welche das Programm enthält und untersucht dieses Zeichen für Zeichen. Zeichenketten werden dabei entweder als Basissymbol erkannt oder aber verworfen. Beziehungen zwischen Basissymbolen werden dazu benutzt eine Baumstruktur aufzubauen, welche die Struktur des Eingabeprogramms wiedergibt. Auf diesem Baum können dann Berechnungen ausgeführt werden deren Ergebnisse ins Zielprogramm ausgegeben werden können. Eli geht deshalb davon aus, daß das eigentliche Übersetzungsproblem aufgeteilt wird in die Probleme des Erkennens der Basissymbole, welche Baumstruktur auf Grund welcher Verbindungen von Basissymbolen aufgestellt werden soll, welche Berechnungen auf diesem Baum ausgeführt werden sollen und wie die Berechnungen für die Ausgabe genutzt werden sollen.

Dieses Kapitel beschreibt nun die Aufteilung des Übersetzungsproblems für die zuvor besprochene Sprache und gibt einen Rahmen an welcher Stelle welche Teilprobleme spezifiziert werden. Die genaue Spezifikation erfolgt dann in den darauf folgenden Kapiteln. Die Abbildung 8.2 zeigt eine feinere Aufsplittung in die Teilprobleme und stellt gleichzeitig den Datenfluß eines Quellprogrammes durch die einzelnen Stationen des Compilers dar [Pad97]. In den Kästchen der Teilprobleme sind die entsprechenden Kapitelnummern dieses Dokumentes vermerkt, in denen genauer auf die Problematik und die Spezifikation eingegangen wird.

8.1 Teilprobleme

Die Aufgabe der *lexikalischen Analyse* ist es das Eingabeprogramm in *Kommentare* und *Basissymbole* zu trennen. Kommentare sind Zeichenketten, die ignoriert werden können, während Basissymbole

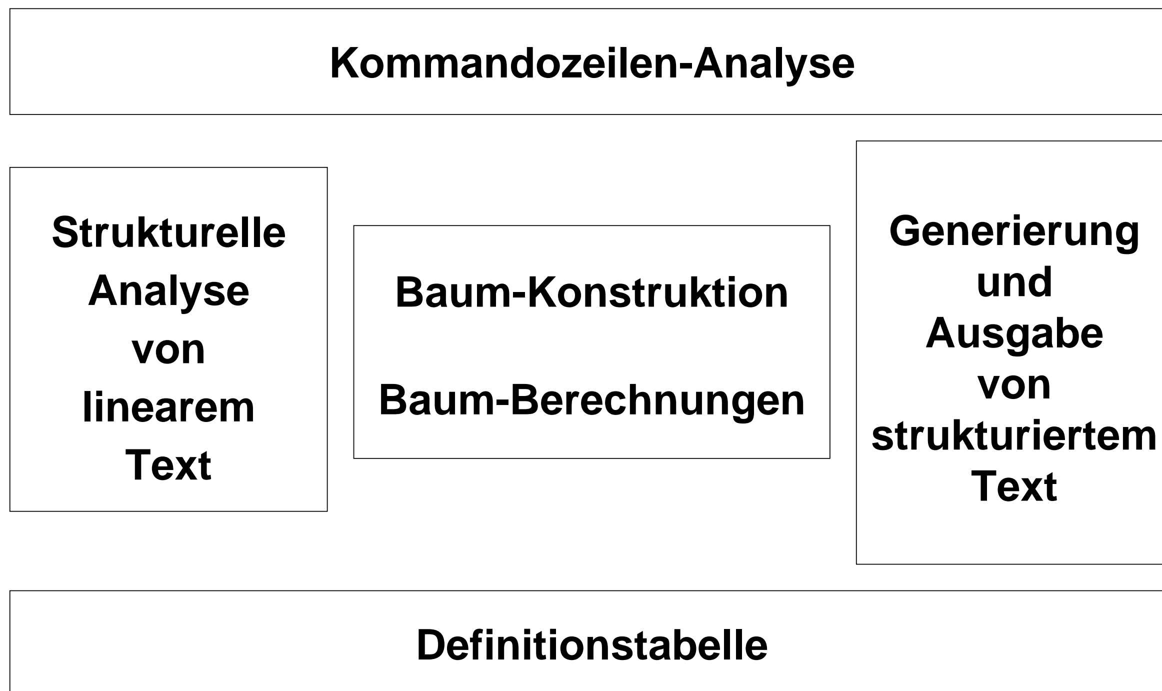

Abbildung 8.1: Grober Aufbau eines typischen Compilers.

Zeichenketten sind, die zu den *Terminalen* der Grammatik gehören und somit die Struktur des Eingabeprogramms definieren. Die Leerzeichen außerhalb von Kommentaren werden dabei herausgefiltert. Jedes Basissymbol wird durch einen Integerwert, dem *Syntax-code* klassifiziert. Jeder *Abgrenzer* besitzt einen einzigartigen Syntax-code und es gibt zu jedem *Identifizierer* (`name`) ebenfalls einen Syntax-code. Darüber hinaus werden Identifizierer auch noch mit einem weiteren Integerwert charakterisiert, dem sogenannten *wahren Attribut* (engl. intrinsic-attribute). Für die Abgrenzer werden keine *wahren Attribute* berechnet. Wenn bei der lexikalischen Analyse ein Basissymbol erkannt wird, so wird die Information, die dieses Symbol charakterisiert, an die *syntaktische Analyse* weitergereicht.

Die *syntaktische Analyse* untersucht die Struktur des Eingabeprogramms. Die Struktur besteht aus einer hierarchischen Zusammensetzung von Phrasen, wobei die Kürzeste das *Basissymbol* ist und die Längste der gesamte *Satz*. Diese Hierarchie kann durch einen Baum wiedergegeben werden, wobei jede Phrase einen Knoten in diesem Baum darstellt. Wenn eine Phrase aus mehreren Phrasen zusammengesetzt ist, so ergeben diese die Kinder des Knotens, der die Zusammensetzung repräsentiert. Basissymbole bilden in einem solchen Baum die Blätter. Die Wurzel des Baumes ergibt den Satz. Bei der Generierung des Baumes wird der *Syntax-code* benutzt, der von der lexikalischen Analyse geliefert wird, um die Basissymbole zu definieren. Wenn Blätter generiert werden, so wird der Wert des *wahren Attributes*, der ebenfalls von der lexikalischen Analyse geliefert wird, in diesem Blatt gespeichert. Abbildung 8.3 veranschaulicht diese Zusammenhänge zwischen Satz, Phrasen und Basissymbolen im Verbindung mit der Baumstrutur.

Wenn der Baum somit einmal aufgebaut ist, so können Berechnungen auf diesem ausgeführt werden, sogenannte *Attributauswertungen* (engl. Attribute-evaluations). Sie bilden meist den Kern eines jeden Compilers und stellen den größten Aufwand und die größte Komplexität dar. Aus diesem Grunde werden solche Attributberechnungen meist in Funktionen unterteilt, um diese besser verständlich zu machen. Die Funktionen, die am häufigsten Verwendung finden, sind die *Bereichsanalyse* (engl.

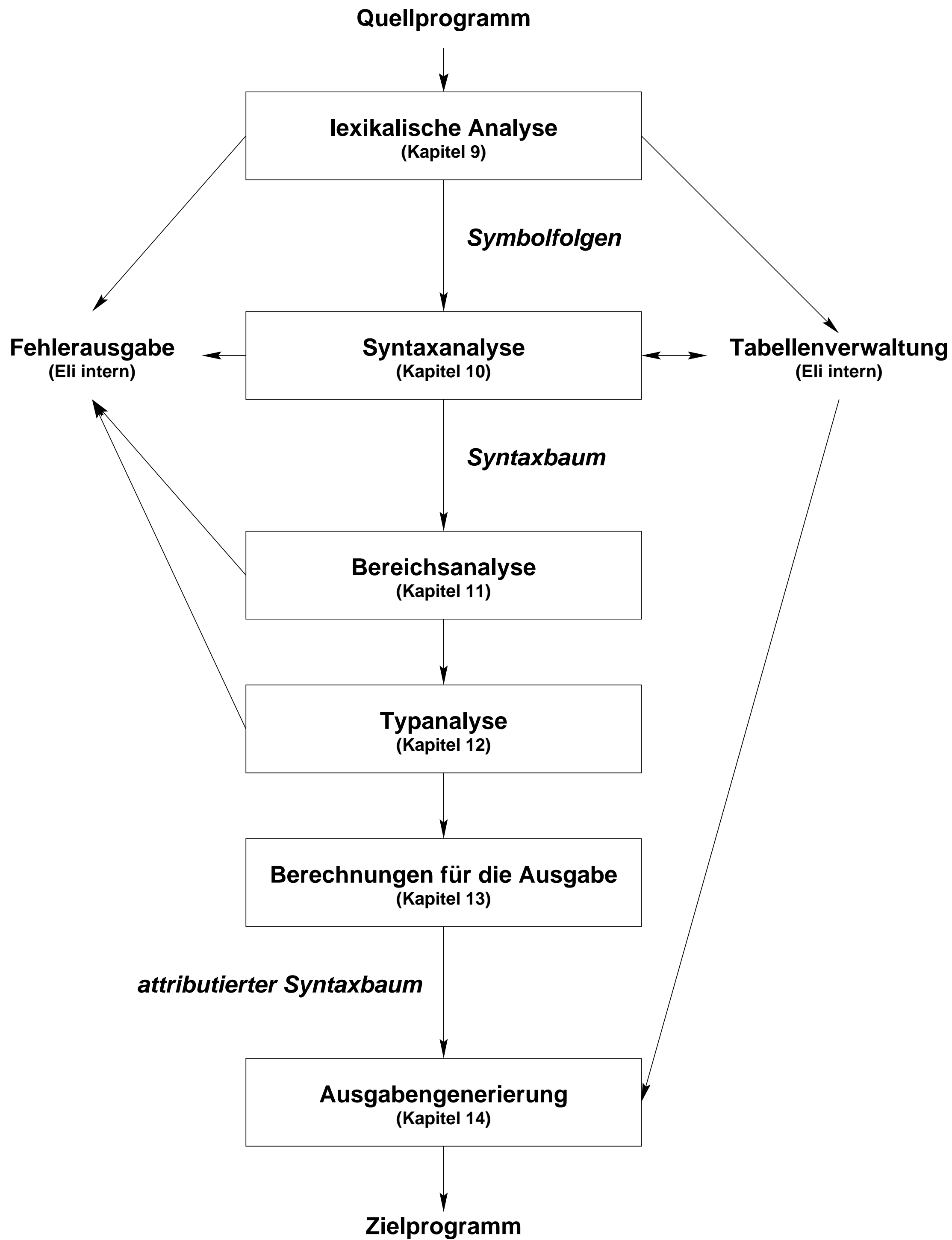

Abbildung 8.2: Der Datenfluß innerhalb eines Compilers, der mit Eli generiert wurde.

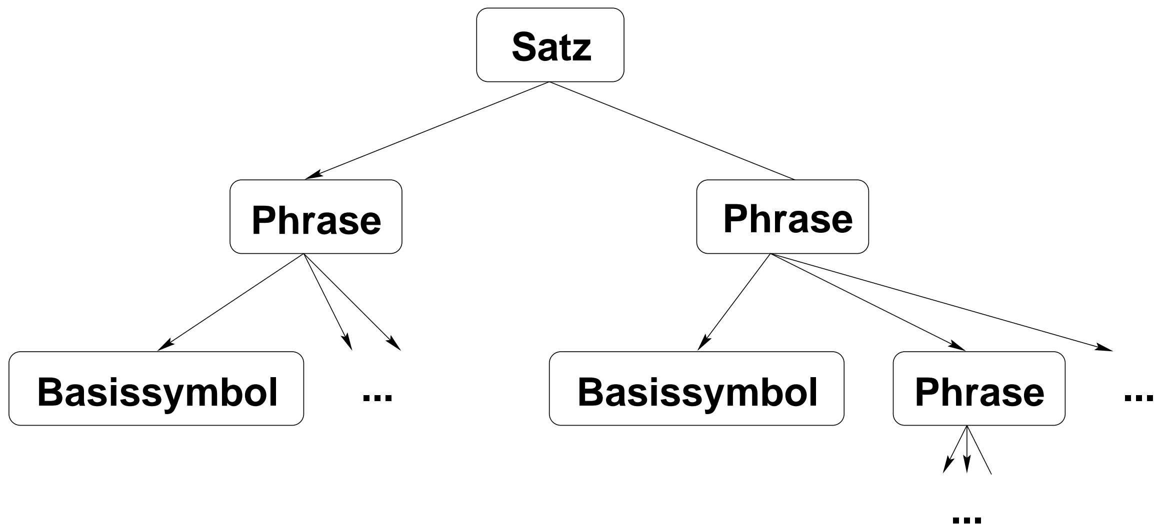

Abbildung 8.3: Baumstruktur der Syntaxanalyse.

Scope-Analysis), welche Entscheidungen für die Verbindung zwischen Benutzung und Definition von Identifizierern trifft, die *Typ-Analyse* (engl. Type-Analysis), die inhaltliche Bedingungen von Ausdrücken überprüft und die *Ausgabengenerierung*, die das Zielprogramm berechnet, eventuell sogar einen kompletten Zielbaum berechnet. Jeder dieser Funktionen ist ein eigenes Kapitel gewidmet (siehe auch Abbildung 8.2).

Der letzte Schritt der Übersetzung ist dann die Ausgabe der Berechnung, also die Linearisierung und Ausgabe des Baumes. Dieser Prozeß wird von Eli durch den *Programm-Text-Generator (PTG)* durchgeführt, welcher mit den Werten aufgerufen wird, die sich während der Attributauswertung ergeben.

Einige Informationen sind mit Zusammenhängen im Baum verknüpft, andere Informationen wiederum sind mit speziellen Instanzen des Programms verbunden. So zum Beispiel besitzen Variablen Typinformationen, die an ihnen heften. Da Variablen aber an verschiedenen Stellen innerhalb eines Programms verwendet werden können, müssen auch deren Typinformationen an diesen Stellen verfügbar sein. Der generierte Compiler beinhaltet für diesen Fall eine *Definitionstabelle* (engl. definition-table), in der solche wichtigen Informationen gespeichert werden können und auf die man über *Schlüssel* (engl. keys) zugreifen kann. Um also z.B. eine Typinformation einer Variablen global verfügbar zu machen, assoziiert der Compiler zu jeder Variablen einen Schlüssel und speichert die Typinformation in der Definitionstabelle über diesen Schlüssel. Dieser Schlüssel ist somit eine eindeutige interne Repräsentierung der Variablen und wird bei jeder Benutzung der Variablen gespeichert.

8.2 Wie ein Compiler spezifiziert wird

Durch die Benutzung von Eli wird der Benutzer gezwungen sich auf die Anforderungen und Entwurfsentscheidungen, die für seinen Compiler wichtig sind, zu konzentrieren und liefert für Probleme, die bei der Compilerentwicklung gängig sind, bereits Lösungen. Eli akzeptiert Beschreibungen solcher Anforderungen und Designentscheidungen und kombiniert diese mit seiner eigenen Auffassung von Compilerbauproblemen, um einen lauffähigen Compiler daraus zu konstruieren. Solche An-

forderungen und Designentscheidungen können als Spezifikationen für die jeweiligen Teilprobleme angesehen werden. Es gibt dabei drei grundlegende Wege, mit denen ein Benutzer seine Teilprobleme spezifizieren kann:

- Durch Analogie ("das Problem ist dasselbe wie Problem X"),
- durch Beschreibung ("das Problem ist ein Problem der Klasse Y und ist wie folgt charakterisiert...") und
- durch Lösung ("hier ist eine Programm, welches das Problem löst").

Bei der Benutzung der *Analogie* stellt Eli eine große Bibliothek von Lösungen zu allgemeinen Teilproblemen beim Compilerbau zur Verfügung. Der Benutzer von Eli muß dabei das Problem so genau verstehen, daß er erkennen kann, daß sein Problem ein allgemeines Problem darstellt und es bereits vorher gelöst worden ist. Für die Verwendung der *Beschreibung* als Problemlösung bietet Eli eine Reihe von Notationen an, um übliche Teilprobleme beim Compilerbau zu spezifizieren. Der Benutzer muß in diesem Fall nicht nur die Art seines Problems erkennen, sondern auch die Notation genau kennen, um sein Problem zu charakterisieren. Die Grammatik aus Abschnitt 6.3, bzw. aus Anhang A ist z.B. eine solche Notation - eine Notation um Phrasenstrukturen einer Sprache zu beschreiben und somit auch das Teilproblem der syntaktischen Analyse zu charakterisieren.

Für die Verwendung von eigenem Programmcode zur Lösung von Teilproblemen akzeptiert Eli normalen C-Code und baut diesen an den entsprechenden Stellen bei der Compilergenerierung einfach ein. Die Verwendung von eigenem Programmcode ist für den Bau dieses Compilers nicht notwendig und es sei an dieser Stelle nur der Vollständigkeit halber erwähnt.

Darüber hinaus stellt Eli ein Paradigma zur Strukturierung der Spezifikationen, das sogenannte *Literate-programming* [Knu92] bereit. Das *Literate-programming* Paradigma schlägt vor, daß Code, oder in diesem Falle Spezifikationen, so zusammengestellt werden sollten, daß es sowohl für den Menschen verständlich wie auch für die Maschine erforderlich ist. Um dieses Problem zu lösen werden in der Regel Dokumente erstellt, die sowohl die einen wie auch die anderen Informationen enthalten. Auf Wunsch können dann sowohl eine für den Menschen verständliche Version als auch eine maschinen-lesbare Version erzeugt werden. Die einfachste Version dieses Paradigmas allerdings erlaubt es einfach dem Benutzer verschiedenste Spezifikationen zu gruppieren. So können Spezifikationen, die eine unterschiedliche Form haben, aber von der Art her zusammengehören, zusammen in einer Datei verarbeitet werden, so daß deren Zusammengehörigkeit klar ist und sie dennoch funktional getrennt behandelt werden können. Von der Möglichkeit Dokumentation und Spezifikationen in einer Datei zu vereinen wurde hier kein Gebrauch gemacht.

8.3 Der eigentliche Compiler

Eli erwartet, daß die komplette Übersetzung in einer einzigen Datei vom Typ `.specs` definiert wird. Diese `.specs`-Datei listet alle Spezifikationen für die einzelnen Teilprobleme auf. Teilprobleme, die durch Analogie gelöst werden können, zeichnen sich durch Referenzierung der entsprechenden Bibliothek aus und Probleme, die durch Beschreibung gelöst werden können, referenzieren eine entsprechende Datei, in der die Charakterisierung des Problems zu finden ist. Die Abbildung 8.4 zeigt die `.specs`-Datei für diesen Compiler.

```
sprache.con
sprache.gla

$/Output/LeafPtg.gnrc :inst
$/Adt/LidoList.gnrc+instance=PTGNode +refer-
to=ptg_gen:inst

name.head
name.lido
schemadef.lido
mappingdef.lido
ausgabe.fw
```

Abbildung 8.4: `.specs`-Datei für den Compiler.

```
1. sprache.specs +arg=(input) :stdout

2. sprache.specs :exe > sprache.exe

3. sprache.specs :source > src
```

Abbildung 8.5: Generierung eines Compilers mittels Eli.

Eine dieser Dateien ist vom Typ `.fw`. Diese folgt dem Paradigma, des Literate-programming. An dieser Stelle wird allerdings davon nur in soweit Gebrauch gemacht, daß zwei einzelne Dateien, die einen semantischen Zusammenhang besitzen, zu einer Datei zusammengefaßt werden.

Die Zeilen, die mit einem $ beginnen, extrahieren Module aus der Eli-Bibliothek, die Problemlösungen durch Analogie anbieten. Jede dieser Module wird an der entsprechenden Stelle in der Dokumentation genau beschrieben.

Um nun aus den von der `sprache.specs`-Datei beschriebenen Spezifikationen einen Compiler von Eli generieren zu lassen, gibt es die im Abbildung 8.5 aufgeführten Möglichkeiten. Die erste Möglichkeit instruiert Eli einen Compiler zu generieren und ihn mit der Commandozeile `input` zu starten. Diese Möglichkeit wird häufig genutzt, um den Compiler schnell mit Beispieleingaben zu testen. Die zweite Möglichkeit generiert einen Compiler und erzeugt eine entsprechend ausführbare `sprache.exe`-Datei im aktuellen Verzeichnis. Diese Version erzeugt einen direkt ausführbaren und von Eli unabhängigen Code. Die letzte Version instruiert Eli in das Verzeichnis `src` alle Sourcecodedateien zu erzeugen, die für die Generierung einer eigenständigen ausführbaren Datei erforderlich sind. Das Verzeichnis `src` enthält die C-Code-Dateien, die Headerdateien, ein Makefile und ein Konfigurationsscript, welches die anderen Dateien an eventuell andere Computer oder Betriebssysteme anpassen kann.

Kapitel 9

Lexikalische Analyse

Die Aufgabe der lexikalischen Analyse ist das Herausfiltern von Wörtern, die das Quellprogramm ausmachen, also denjenige Aneinanderreihungen von *Basissymbolen*, die zuvor in der Grammatik als *wichtig* definiert worden sind. Dabei werden Vokabelfehler, wie z.B. unzulässige Buchstaben, fehlende Leerzeichen oder falsch gekennzeichnete Kommentare erkannt und als Fehler während der lexikalischen Analyse ausgegeben. Ein Compiler, der von Eli generiert worden ist, generiert auch eine solche Kette von Basissymbolen, wenn Vokabelfehler gefunden worden sind. Die so produzierte Sequenz beinhaltet dann nur die Basissymbole, die korrekt bezüglich der Sprachendefinition sind und gibt somit das Programm im Hinblick auf die Fehler so genau wie möglich wieder.

Der Prozeß der lexikalischen Analyse untersucht den Eingabetext Zeichen für Zeichen von links nach rechts. Er erkennt dabei die Basissymbole und verwirft die Trennzeichen und Kommentare. Er ermittelt für jedes Basissymbol einen *Syntax-Code* und für jeden Bezeichner das sogenannte *wahre Attribut*.

Für das Einlesen des Quelltextes stellt Eli ein eigenständiges Modul bereit, welches den Quelltext dann auch für die folgenden Module zur Verfügung stellt. Dieses Modul braucht nicht weiter spezifiziert werden. Es wird von Eli, sowie es gebraucht wird, selbstständig aus der Bibliothek abgeleitet. Im Anhang B.1 auf Seite 91 wird die Vorgehensweise dieses Moduls einführend erläutert.

Der Prozeß der lexikalischen Analyse baut auf dem so eingelesenen Text ein Koordinatensystem auf, wobei jede Koordinate aus einer Zeilennummer und einer Spaltennummer innerhalb dieser Zeile besteht. Diese Angaben werden für die verschiedenen Komponenten, die auf dem Text arbeiten, exportiert.

Der Prozeß der *lexikalischen Analyse* und der Prozeß der *syntaktischen Analyse* arbeiten eng zusammen. Diese Zusammenarbeit wird über den Zwischencode realisiert. Hierbei spielen die schon vorher erwähnten *Syntax-Codes* und *wahren Attribute* eine Rolle. Da dies jedoch von Eli intern verarbeitet wird, sei an dieser Stelle auf den Anhang B.2 auf Seite 91 verwiesen.

Der eigentliche Vorgang des Scannens basiert nun auf der Unterscheidung zwischen *Literalen* und *Nichtliteralen*. In Abschnitt 6.2 wurden bereits die in der Sprache verwendeten *Abgrenzer* vorgestellt und sind durch Grammatik der Sprache bereits eindeutig spezifiziert. In der Sprache werden aber auch drei verschiedene Nichtliterale als Basissymbole verwendet, `name`, `schema_type_spec` und `simple_type_spec`. Diese müssen für den Vorgang der lexikalischen Analyse jedoch näher spezifiziert werden. Für die zugelassenen Kommentare gilt dasselbe. Auch diese müssen genau spezifiziert werden, da sie sich natürlich nicht aus der Grammatik ableiten lassen. Da in den meisten Programmiersprachen nur wenig unterschiedliche Arten von Basissymbolen und von Kommentaren vorkom-

men, stellt Eli für ihre Spezifikation bereits Bibliotheken zur Verfügung. Darüber hinaus werden mit den Bibliotheken auch die notwendigen Methoden für die Generierung der *wahren Attribute* mitgeliefert. Eine genaue Erklärung dieser Spezifikation und die Funktionsweise der Bibliotheken sowie die damit verbundenen Besonderheiten findet sich im Anhang B.3 ab Seite 92. Dieses Teilproblem kann demnach teilweise durch Analogie gelöst werden.

Als ein Ergebnis des Scanvorganges wird die sogenannte *Identifizierer-Tabelle* (engl. identifier-table) aufgebaut. Sie ist für die Verwaltung der im Quelltext vorkommenden Bezeichner und die dazugehörigen *wahren Attribute* zuständig. Unter Zuhilfenahme der zuvor erwähnten Bibliotheken und Methoden wird diese Tabelle von Eli selbstständig aufgebaut. Aus diesem Grund sei für die zugrundeliegende Vorgehensweise auf Anhang B.4 auf Seite 93 verwiesen.

Die sich aus diesem Abschnitt ergebenden Spezifikationen werden in Eli in der Datei `sprache.gla` zusammengestellt. Der Inhalt dieser Datei findet sich im Anhang B.5 auf Seite 93.

Kapitel 10

Syntaktische Analyse

Die Aufgabe der syntaktischen Analyse ist es, die Struktur des Quellprogramms zu analysieren. Dies geschieht, indem die hierachische Struktur der einzelnen Phrasen erkannt wird und ein Syntaxbaum aufgebaut wird, der für jede dieser Phrasen einen Knoten enthält. Während dieser Analyse werden Strukturfehler, wie z.B. nicht korrekt geschlossene Blöcke oder mit anderen Worten nicht korrekte Klammerungen, aufgespürt und als Fehlermeldung ausgegeben. Ein von Eli generierter Compiler baut auch dann noch einen solchen Syntaxbaum auf, wenn strukturelle Fehler gefunden werden. Die sich daraus ergebende Baumstruktur ist bezüglich der Sprachdefinition korrekt und gibt das Quellprogramm mit Rücksicht auf die Fehler so genau wie möglich wieder.

Die Grundlage für die Spezifikation der syntaktischen Analyse liefert die Grammatik der Sprache. Sie spezifiziert die Struktur der Phrasen für jedes Quellprogramm. Es ist meistens notwendig die Grammatik zu modifizieren, um sicherzustellen, daß sie vollständig und eindeutig ist. Um dieses zu gewährleisten kann es z.B. notwendig sein manche Phrasen zu unterscheiden, die dieselbe Bedeutung haben und somit auch dieselbe Art von Knoten im Baum haben sollten. Diese Punkte sind demnach Gegenstand dieses Abschnittes. Für die dazugehörenden Spezifikationen wird auf den entsprechenden Anhang verwiesen.

Aus der zuvor genannten Aufgabenstellung für den Syntaxanalysierer geht hervor, daß sich dieser direkt aus der Grammatik ableiten lassen muß. Da Eli allerdings eine spezielle Form eines Syntaxanalysierers generiert, müssen zunächst gewisse Vorbedingungen geschaffen werden, die die Grammatik betreffen. Dies bedeutet, daß der erste Entwurf der Grammatik angepaßt werden muß.

Eli generiert einen sogenannten LALR(1)-Parser. Dieser kann nur aus der Grammatik abgeleitet werden, wenn die folgende Bedingung von der Grammatik erfüllt wird: Zu jedem Zeitpunkt des Durchlaufes durch das Quellprogramm von links nach recht muß der Parser in der Lage sein entscheiden zu können, ob er sich am Ende einer Phrase befindet oder nicht und wenn ja, welche Phrase gerade beendet worden ist.

Die Grammatik aus Kapitel 6.3 erfüllt diese LALR(1)-Bedingungen, allerdings stimmt sie nicht ganz mit der Definition der Vokabeln aus Kapitel 6.2 überein. In der Grammatik werden u.a. verschiedene Arten von `names`, wie z.B. `type_name_def`, `type_name_use` oder `attribute_name_def`, verwendet, bzw. in der für Eli verwendeten Grammatik umbenannt, weil dies für spätere Problemlösungen von Wichtigkeit ist. Diese und andere Symbole sind weder in der Grammatik genauer definiert, noch sind sie als Basissymbole definiert. Der menschliche Leser wird sicherlich keine Schwierigkeiten damit haben anzunehmen, daß es sich dabei in allen Fällen um `names` handelt, denn die Namensgebung ist semantisch behaftet; dieser Zusammenhang muß allerdings explizit gemacht

```
schema_name_def : name .
type_name_use : name .
type_name_def : name .
...
```

Abbildung 10.1: Redefinition von `names` in verschiedenen Kontexten.

werden, wenn die Grammatik den Parsingprozeß spezifizieren soll. All diese Symbole definieren einen ganz bestimmten Zusammenhang in der Grammatik, der die Bedeutung des names beschreibt. Diese unterschiedliche Bedeutung muß auch im Syntaxbaum wiedergegeben werden, in dem es sich auch dort um unterschiedliche Knoten handelt. Einige der notwendigen Umbenennungen, also Erweiterungen in der Grammtik sind in Abbildung 10.1 aufgeführt. Die Auflistung aller Ergänzungen findet sich in Anhang C.1 auf Seite 95.

Nach der Erweiterung der Grammatik kann der Parser für die syntaktische Analyse direkt aus der Grammtik abgeleitet werden.

Die sich aus diesem Kapitel ergebenden Spezifikationen sind im Anhang C.2 auf Seite 96 erläutert. Die vollständige Grammatik in Eli-Notation mit all ihren notwendigen Änderungen gegenüber der BNF-Version aus Anhang A ist im Anhang C.3 auf Seite 96 aufgeführt.

Kapitel 11

Bereichsanalyse

Um auf Variablen in einem Quellprogramm dieser Sprache zu verweisen, werden Namen benutzt. Jeder dieser Namen muß im Definitionsteil des Quellprogramms definiert werden und kann dann später an einer beliebigen Stelle innerhalb einer bestimmten Umgebung oder Bereiches (engl. scope) benutzt werden. Eine Umgebung einer Definition ist eine Phrase eines Programmes und innerhalb dieser Phrase darf für einen Namen nur einmal eine Definition erfolgen. Die Aufgabe der Bereichs-Analyse ist es demnach sicherzustellen, daß es für jeden Namen nur genau eine Definition gibt und einen Zusammenhang jeder Benutzung des Namens und seiner Definition herzustellen.

Da in dieser Sprache keinerlei gleichartige Blöcke ineinander verschachtelt werden, also z.B. innerhalb von Klassen keine weiteren Klassen definiert werden, entfällt ein sehr großer Aufgabebereich der geschachtelten Bereichsanalyse. Die Aufgabe der Bereichsanalyse für die hier verwendete Sprache ist demnach auf zwei Aufgaben zu beschränken:

- Innerhalb von gewissen Umgebungen müssen Namen eindeutig sein. Dies bedeutet, daß ihre Definition nur einmal erfolgen darf. In dieser Grammatik bedeutet dies z.B., daß Attribute, die innerhalb einer Klasse definiert werden, eindeutig sind. Dies betrifft genauso die Definition von Klassen, Pointern und Listen innerhalb von Schemata. Die Schemata selbst müssen ebenfalls eindeutig sein. Dies gilt es zu prüfen.

- Alle diese Definitionen von Namen müssen vor ihrer Verwendung erfolgen, d.h. die Verwendung eines nicht definierten Namens ist unzulässig und muß eine Fehlermeldung hervorrufen.

Für die Eindeutigkeit von Namen innerhalb von Umgebungen ist also nur wichtig zu wissen welche Symbole in der Grammtik eine solche Umgebung definieren und welche Symbole darin eindeutig zu sein haben. Eli stellt für die Lösung dieses Problems zwei Module zur Verfügung, `Nest` und `Unique`. Das Modul `Nest` stellt dabei das Konzept für geschachtelte Bereiche zur Verfügung, welches auch auf einfache, ungeschachtelte Bereiche anwendbar ist. Das Modul `Unique` hingegen stellt das Konzept der Fehler für Eindeutigkeiten innerhalb gewisser Bereiche zur Verfügung. Somit können diejenigen Symbole der Grammatik benannt werden, die eindeutig in einem Bereich sein müssen. Anschließend müssen dann noch diejenigen Symbole, die einen Bereich darstellen, mit entsprechenden Bereichsdefinitionen versehen werden. Hierbei muß eine alles umschließende Hauptumgebung definiert werden. In ihr werden dann ggf. weitere Umgebungen definiert. Bei den daraus resultierenden Überprüfungen sind dann die zu kontrollierenden Namenssymbole von Umgebungen umschlossen und werden auf Eindeutigkeit überwacht.

Für die Überprüfung von Definition und Verwendung von Namen ist eine Vorraussetzung, daß diese Symbole in der Grammatik namentlich getrennt sind. Dies ist auch in der hier verwendeten Grammatik der Fall, z.B. wird in die Symbole `type_name_def` und `type_name_use` unterschieden. Das Modul `AlgScope` von Eli stellt einige Methoden zur Verfügung, um dieses Problem zu lösen. Das Grundprinzip basiert darauf, daß für alle Namen (Bezeichner) im Quelltext ein Attribut namens `Sym` berechnet wird, welches den verwendeten Namen in Form einer Integer-Zahl wiedergibt. Diese Integerzahl ist für eine bestimmte Zeichenkette eindeutig. Unterschiedliche Namen führen demnach zu unterschiedlichen Zahlen und gleiche Namen ergeben denselben Zahlenwert. Das Modul berechnet nun den Wert eines `Key`-Attributes in jedem Knoten des Syntaxbaumes, der einer Definition oder Benutzung eines Namens entspricht. Der Wert dieses `Key`-Attributes wird bei einer Verknüpfung von Defintion und Benutzung gleich sein. Wenn jedoch eine Benutzung nicht zuvor mit einer Definition verknüpft sein sollte, so wird dessen `Key`-Attribut den Wert "`NoKey`" aufweisen. Dieser Sachverhalt kann dann sehr einfach zur Generierung einer Fehlermeldung genutzt werden.

Diese Funktionalität ist in der aktuellen *Beta*-Version des Compilers nicht implementiert. Hauptaugenmerk war zunächst die Implementierung eines lauffähigen Compilers, bevor auf zusätzliche Funktionalität geachtet wurde.

Kapitel 12

Typ-Analyse

Die Aufgabe der Typ-Analyse ist es festzustellen welche Art von Objekt von einem Namen repräsentiert wird und daß in einem bestimmten Kontext in einem Quellprogramm nur Namen verwendet werden, die die richtigen Objekte repräsentieren. Wenn also z.B. bei der Verwendung eines Attributnamens ein Klassenname angegeben wird, so führt dies zu einer Fehlermeldung.

Die Typ-Analyse ist ein Prozeß, der sich sehr stark auf die Verwendung von Attributen und Berechnungen auf diesen abstützt. Es kann jedoch eine allgemeine Vorgehensweise skizziert werden:

In Zusammenhang mit den `Key`-Attributen aus dem vorhergehenden Kapitel 11 der Bereichsanalyse werden für jedes betroffene Symbol *Art*-Eigenschaften eingeführt. Dazu sind zunächst alle möglichen *Arten* von Typen zusammenzutragen. Diese werden dann z.B. durch *ganze Zahlenwerte* repräsentiert. Von Eli generierte Compiler speichern solche Informationen über Symbole in der *Definitionstabelle*. Es handelt sich dabei um eine globale Datenbank, welche über die *Keys*, die mit den Symbolen assoziiert wurden, benutzt werden kann. Diese Eigenschaften können von beliebigem Wert sein und werden in diesem Fall durch ganze Zahlen beschrieben. Zusätzlich zu den *Arten* müssen noch weitere spezifische Funktionen definiert werden, die den Zugriff auf diese *Arten* ermöglichen. Darüber hinaus kann dann ein Attribut namens `Type` für jeden betroffenen Knoten spezifiziert, welches in der Lage ist einen Rückgabewert mit Informationen zu liefern.

Auf diese Art und Weise wird es möglich, Namen bei ihrer Definition gewisse Eigenschaften anzuhängen und sogar während des Übersetzungsvorganges zu verändern oder auch zu ergänzen und diese zu geeigneter Zeit bei der Verwendung zu überprüfen.

In der aktuellen *Beta*-Version des Compilers wurde auf diese Funktionalität aus Komplexitätsgründen verzichtet. Das Hauptaugenmerk lag zunächst auf der Implementierung eines lauffähigen Compilers, bevor auf komplexe, zusätzliche Funktionalität geachtet wurde.

Kapitel 13

Die Ausgabemuster

Ein Quellprogramm, welches in der Spezifikationsprache geschrieben ist, definiert in seiner Weise eine ganz bestimmte Datenstruktur. Mit Hilfe des Compilers werden diese Definitionen ohne Informationsverlust in einen *C*++-Code überführt, der später in der Lage sein wird die beschriebene Datenstruktur nutzbar zu machen. Dieser Abschnitt beschreibt die einzelnen Komponenten des auszugebenden *C*++-Codes und wie diese mit Hilfe von deklarativer Spezifikationen beschrieben werden können.

In Eli kann zur Ausgabe eines strukturierten Textes ein eigener Generator, der Pattern-Based Text Generator (kurz: PTG), benutzt werden. Die Struktur des Zieltextes wird hierbei durch eine Menge von Mustern beschrieben. PTG generiert daraus für jedes Muster eine eigene Funktion. Diese können dann aufgerufen werden, um eine Instanz der Zielstruktur zu konstruieren, die schließlich ausgegeben werden kann. PTG kann dazu benutzt werden jede Art von Zielsprache zu generieren, also z.B. Programme beliebiger Programmiersprachen, spezielle Sprachen wie LaTeX oder Postscript oder auch einfach nur strukturierte Daten in Textform. Letzteres wird für diesen Compiler verwendet. Die von PTG generierten Funktionen werden in der Regel in LIDO-Berechnungen verwendet (also in Berechnungen, die auf Knoten im Syntaxbaum definiert sind), um für bestimmte Symbole entsprechende Ausgabetexte zu konstruieren.

Eine PTG Spezifikation ist eine Menge von Mustern, die mit Namen versehen werden, welche die Struktur und die Inhalte der einzelnen Komponenten der Ausgabe beschreiben. Aufrufe der daraus entstandenen Funktionen erzeugen eine Instanz dieser Struktur, welche sich auch durch eine Baumstruktur darstellen läßt. Die einzelnen Knoten dieses Baumes sind dann von dem PTG-eigenen Typ `PTGNode`. Zur Ausgabe dieser Struktur wird die Wurzel des Baumes an eine PTG-Ausgabefunktion übergeben. Die Struktur wird darauf hin linearisiert als Text ausgegeben.

Die Abbildung 13.1 gibt einen Ausschnitt der Struktur des Ausgabetextes wieder. Aus ihr wird die hierarchische Struktur der Ausgabe und auch die Verbundenheit zur Eingabesprache ersichtlich. Bei den einzelnen Kästen handelt es sich um die zuvor erwähnten, benannten Muster, die sich wiederum aus Teilmustern zusammensetzen können. Die Wurzel bildet hierbei das Muster `Table`, welches aus den Mustern `SchemaList` und `PathName` zusammengesetzt ist. Dies spiegelt die Vorgehensweise in der Spezifikationssprache wieder. Eine Spezifikation setzt sich aus dem Definitionsteil und dem Mappingteil zusammen. Muster, die mit einem “*” versehen sind, können mehrfach auftreten. Für jedes der aufgeführten Kästen muß nun ein Muster definiert werden, welches beim Erkennen bestimmter Symbole zu der entsprechenden Textausgabe führt. Als Beispiel für die Spezifikation eines solchen Musters sei hier das Muster `SchemaList` aufgeführt. Es wird in Abbildung 13.2 gezeigt. Hierbei wird die Struktur eines solchen Musters deutlich. Die vorkommenden $-Zeichen stehen für

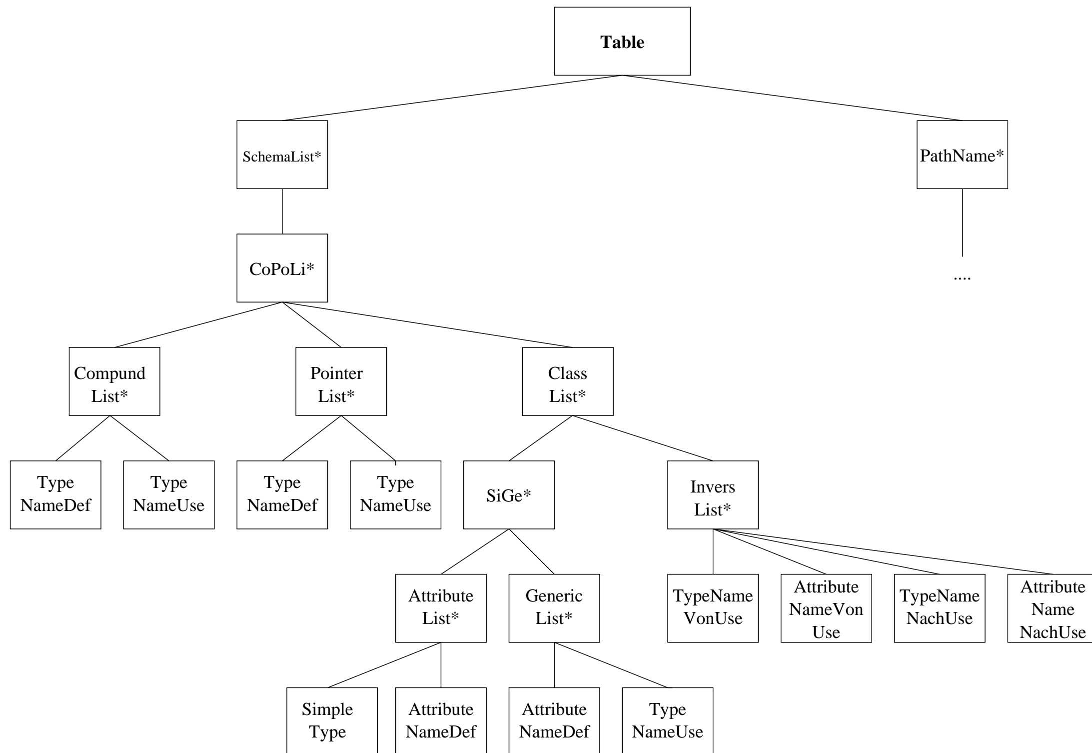

Abbildung 13.1: Die Struktur des Ausgabetextes.

Übergabeparameter zu diesen Mustern, bzw. für die daraus resultierenden Funktionen und Angaben in Anführungszeichen werden direkt ausgegeben.

Eine ausführliche Erläuterung zur Spezifikation der Muster und die Zusammenhänge der Muster untereinander sowie die Einordnung der Ausgaben in den Kontext der Spezifikation findet sich im Anhang D auf den Seiten 99 bis 108. Die daraus resultierenden Spezifikationsdateien für Eli sind in Anhang D.4 auf Seite 108 aufgeführt.

```
SchemaList:
        /* $1 = Schemaname
           $2 = Liste der Klassendefinitionen
           $3 = SchemaTyp */

        "// naechstes Schema\n\n"
        "PCSchema " $3 "_" $1 "=new CSchema( \"" $3 $1 "\",pg);\n\n\n"
        "// Schema mit Klassen fuellen\n\n\n"
        "//Typen\n"
        "BuildSimpleTypes ( " $3 "_" $1 " );\n\n"
        $2 /* Liste der Klassendefinitionen */
        "// Ende der Klassen\n"
```

Abbildung 13.2: Das Ausgabemuster SchemaList.

Kapitel 14

Generierung der Ausgabe

Die Aufgabe der Codegenerierung ist es die Konzepte der Quellsprache in die Konzepte der Zielsprache zu überführen. Während dieses Prozesses sollten keine Fehler mehr gefunden oder ausgegeben werden.

Die Codegenerierung ist ein Attributierungsprozeß, d.h. es werden Berechnungen auf Knoten des Syntaxbaumes ausgeführt. Für alle Blätter des Baumes, die für die Ausgabe relevant sind, werden dabei z.B. ausgabefähige Attribute generiert, eventuell zusätzlich benötigte Informationen generiert und in die entsprechenden Knoten kopiert, für die unterschiedlichsten Knoten die entsprechenden Ausgaben konstruiert und diese Informationen letztendlich ausgegeben.

14.1 Die Ausgabe

Jede Anweisung, die vom Compiler ausgegeben wird, basiert auf einer bestimmten Berechnung mit keinem oder mehreren Operanden. Diese einzelnen Berechnungsteile werden durch die Muster aus Kapitel 13 zur Verfügung gestellt. Jedes Muster besitzt dabei einen eigenen Namen und der Code, der diesem Muster entspricht, wird generiert, indem eine Funktion aufgerufen wird, deren Name mit PTG beginnt und von dem Musternamen gefolgt wird. So wird also z.B. der Code für das Muster `SchemaList (Schemaname, Klassenliste, Schematyp)` generiert, in dem die Funktion `PTGSchemaList(Schemaname, Klassenliste, Schematyp)` aufgerufen wird. Der durch diesen Aufruf zurückgelieferte Code ist vom Typ `PTGNode` und kann als Argument für weitere Funktionsaufrufe dienen.

Der so generierte Code wird aber durch die Funktionsaufrufe noch nicht ausgegeben. Vielmehr wird eine explizite Datenstruktur aufgebaut, die den generierten Code enthält. Diese Datenstruktur entspricht einer Baumstruktur und kann ausgegeben werden, in dem die Wurzel des Baumes an die Funktion `PTGOut` übergeben wird. In diesem Falle wird für das Symbol `specification`, dem obersten Symbol in der Grammatik, ein entsprechender Code generiert und in dem Attribut mit dem Namen `Ptg` gespeichert. Die Ausgabe erfolgt demnach durch Übergabe dieses Attributes an `PTGOut`:

```
SYMBOL specification COMPUTE
  PTGOut( THIS.Ptg )
END;
```

14.2 Generierung ausgabefähiger Attribute

Für die Generierung von Ausgaben mit Hilfe des PTG müssen die entsprechenden Ausgaben in Form von PTGNodes vorliegen. Die Attribute im Syntaxbaum, der von Eli generiert worden ist, liegen allerdings nicht in einem solchen Format vor und müssen deshalb umgewandelt werden. Da diese Umwandlungen für viele verschiedene Symbole der Grammatik durchgeführt werden müssen, stellt Eli die Möglichkeit zur Verfügung ein generisches Symbol zu definieren (Entity), auf welchem die Berechnungen spezifiziert werden können. Später können dann die Symbole angegeben werden, auf denen die Berechnungen des generischen Symbols ausgeführt werden sollen. Die Attribute, auf denen diese Berechnungen ausgeführt werden sollen, sind alle `names`, wie durch die Grammatik beschrieben und demzufolge würde beispielhaft die Spezifikation eines solchen generischen Symbols wie folgt aussehen:

```
ATTR Sym: int;
SYMBOL Entity COMPUTE
  SYNT.Sym = CONSTITUENT name.Sym;
END;
ATTR Ptg: PTGNode;
SYMBOL Entity INHERITS IdPtg END;
```

Eine intensive Betrachtung der hierfür verwendeten Module und Befehle und deren Auswirkungen erfolgt in Anhang E.1 ab der Seite 109. An dieser Stelle sei nur wichtig, daß durch Anwendung dieser Berechnungen auf ein Symbol der Grammatik, welches einen `name` enthält ein ausgabefähiges Attribut mit dem Namen `Ptg` generiert werden kann. Demzufolge werden diese Berechnungen auf alle Symbole der Grammatik angewendet, die einen `name` beinhalten und die für die spätere Ausgabe von Wichtigkeit sind. Die dafür notwendige Vorgehensweise wird ebenfalls im Anhang E.1 ausführlich geschildert.

Da jedoch in der Grammtik nicht ausschließlich Symbole verwendet werden, die einen `name` beinhalten, sondern darüber hinaus auch noch `simple_type_spec` und `schema_type_spec` Verwendung finden, sei darauf hingewiesen, daß ähnliche Berechnungen auch für die Symbole definiert werden müssen, die diese Terminal-Attribute verwenden, da auch diese für die spätere Ausgabe benötigt werden (siehe ebenfalls Anhang E.1).

Als Ergebnis dieser Berechnungen existieren nun zu allen Terminal-Attributen, die für den Ausgabetext relevant sind, die dafür notwendigen Ptg-Attribute in den jeweiligen Knoten des Syntaxbaumes.

14.3 Berechnungen für den Definitionsblock

Aus Kapitel 13 ist bereits bekannt, daß für die Ausgabe einer Klassendefinition der Schemaname und der Schematyp erforderlich sind, denn dies sind u.a. zwei Parameter für den Aufruf der Musterfunktion `PTGClassList`. Aus der Grammtik wird allerdings deutlich, daß die Funktion `PTGSchemaList` jedesmal dann ausgeführt werden muß, wenn das Symbol `complex_decl` der Grammatik erkannt wird. In dem dazugehörigen Syntaxbaum liegen die Informationen über den Schematyp und den Schemanamen allerdings in Knoten, die parallel zu dem Zweig von `complex_decl` liegen, nämlich in dem Zweig von `schema_specifier`. Da während der Berechnungen auf Knoten nicht auf solche parallel verlaufenden Zweige zugegriffen werden kann, müssen die Daten für den Schemanamen und den Schematyp zunächst in die gemeinsame Wurzel kopiert werden. Dann ist es in

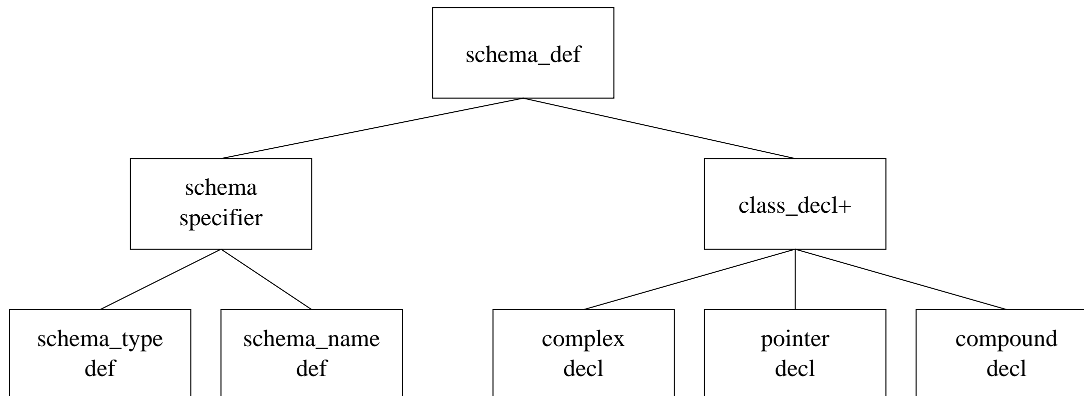

Abbildung 14.1: Kleiner, struktureller Auszug aus dem Syntaxbaum.

späteren Berechnungen möglich diese Daten in direkt unterhalb dieser Wurzel liegenden Knoten zu benutzen oder bei Bedarf auch in diese Knoten zu kopieren.

Abbildung 14.1 verdeutlicht diese Problematik.

Um nun Attribute aus den Blättern `schema_type_def` und `schema_name_def` in den Knoten `complex_decl` zu bekommen, da sie dort benötigt werden, wird folgendermaßen vorgegangen:

1. Im Syntaxbaum muß zunächst die für beide Knoten gemeinsame Wurzel ausfindig gemacht werden.
2. Die Attribute, die von Interesse sind, werden in der Wurzel neu angelegt und aus den Quellknoten *heraufkopiert*.
3. Damit diese Attribute nun bei der späteren Verwendung leichter zu erreichen sind, werden dieselben Attribute im gewünschten Zielknoten ebenfalls neu angelegt und anschließend dorthin *hinunterkopiert*.

In dem hier aufgeführten Falle würde dies das jeweils ausgabefähige PTG-Attribute von `schema_type_def` und `schema_name_def` betreffen. Diese würden in die Wurzel `schema_def` kopiert und anschließend in den Knoten `complex_decl` weitergeleitet. Die Abbildung 14.2 zeigt diese Vorgehensweise anschaulich. Für diesen Kopiervorgang ist es übrigens, wie auch in der Abbildung zu sehen, uninteressant wieviele Knoten auf dem *Weg* liegen.

Die für diesen Kopiervorgang notwendigen Spezifikationen in Eli und die verwendeten Module sowie eine ausführliche Erläuterung der verwendeten Spezifikationskonstrukte und deren Funktion können im Anhang E.2 ab Seite 111 nachgelesen werden.

Für den Definitionsblock sind dies jedoch nicht die einzigen Kopiervorgänge, die notwendig sind. Für die Ausgaben von Attributdefinitionen ist neben dem Schemanamen und Schematyp auch noch der zugehörige Klassenname notwendig (siehe Muster `PTGAttributeList` in Anhang D.2.4, Seite 103). Aus der Grammtik läßt sich allerdings erkennen, daß sich das Symbol `simple_decl` aus dem Symbol `complex_decl` verzweigt. Der zugehörige Klassenname verzweigt sich auch hier im Syntaxbaum von demselben Knoten `complex_decl`. Somit tritt auch hier der Fall ein, daß bei den Berechnungen auf dem Knoten `simple_decl` nicht auf den Klassennamen, der sich in `type_name_use` befindet und in einem parallelen Zweig liegt, zugegriffen werden kann. Abbildung 14.3 zeigt dieses Sachverhalt.

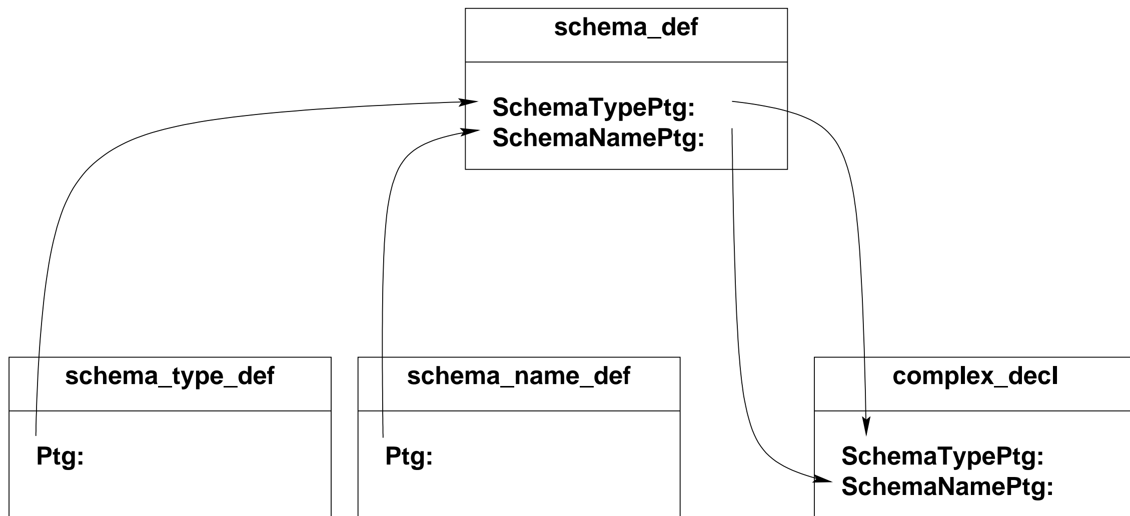

Abbildung 14.2: Vorgehensweise beim Kopieren von Attributinhalten in parallel verlaufende Zweige im Syntaxbaum.

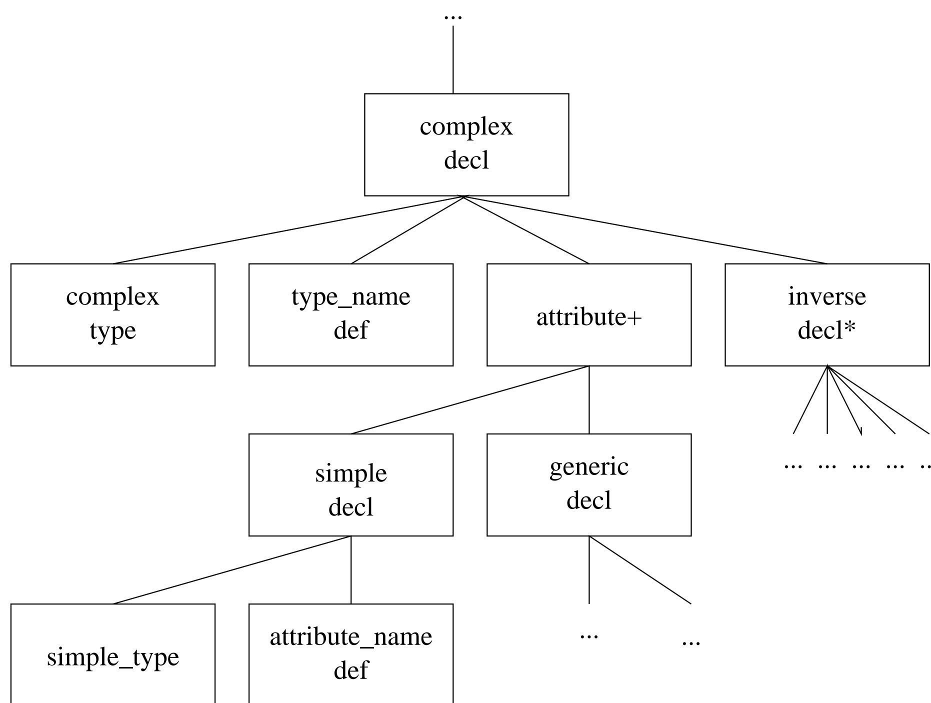

Abbildung 14.3: Ausschnitt aus dem Syntaxbaum.

Aus diesem Grunde muß auch hier zunächst der Klassenname in den gemeinsamen Knoten `complex_decl` kopiert werden. Um später einfacher auf die Daten zugreifen zu können werden nun der Schemaname, der Schematyp und der Klassenname direkt in den Knoten `simple_decl` kopiert. Die Vorgehensweise ist analog zu der vorhin beschriebenen und wird im Anhang E.2 detailiert beschrieben.

Neben dem Symbol `simple_decl` existiert auch noch ein weiteres Symbol, welches Attribute im Quelltext definieren kann: `generic_decl`. Da für dieses Symbol dieselben Informationen benötigt werden, um eine entsprechende Ausgabe zu generieren, werden auch in diesen Knoten die notwendigen Daten kopiert.

Neben Klassen können in Schemata auch noch Pointer, Listen und inverse Attribute definiert werden (Abbildung 14.1). Für eine Ausgabe einer Pointerdefinition werden neben den direkt angegebenen Klassen- und Attributnamen auch wieder der Schematyp und der Schemaname benötigt, die im Syntaxbaum oberhalb von `pointer_decl` liegen. Da dieser Zweig parallel zu dem Zweig der Klassendefinition (`complex_decl`) liegt, liegen die notwendigen Daten ja bereits im Knoten `schema_def` bereit und können deshalb für die einfachere Benutzung direkt in den Knoten `pointer_decl` kopiert werden.

Eine analoge Symbolberechnung findet demnach auch für die Listendefinition und für die inversen Attribute statt, also auf dem Knoten `compound_decl` und `inverse_decl` im Syntaxbaum.

Somit sind für den Definitionsblock die Berechnungen in soweit abgeschlossen, als daß in den zur Ausgabe führenden Knoten `complex_decl`, `pointer_decl`, `compound_decl`, `inverse_decl`, `simple_decl` und `generic_decl` die jeweils notwendigen Daten vorhanden sind, bzw. leicht auf die noch fehlenden zugegriffen werden kann.

14.4 Berechnungen für den Mappingblock

Bei den Angaben im Mappingblock der Sprache sind die Definitionen stärker ineinander verschachtelt. Auch in der Grammatik sind die verschiedenen Symbolnamen difizieler benannt, damit diese bei späteren Knotenberechnungen genau auseinander gehalten werden können. Durch die stärkere Verschachtelung wird es notwendig sein entsprechende Daten häufiger im Syntaxbaum zu kopieren, um sie an den verschiedensten Stellen nutzbar zu machen und auch deren Zugriff zu vereinfachen. Durch die Verwendung vieler unterschiedlicher Symbole unterhalb eines Knotens wird es auch notwendig sein sehr viele Attribute zu kopieren, weil auch für die Generierung der entsprechenden Ausgaben soviele Daten erforderlich sind.

Für die Ausgaben sind in allen Fällen jeweils der Schemaname und der Schematyp erforderlich. Diese werden in der Grammatik an vielen Stellen durch ein eigenes Symbol spezifiziert, was im Syntaxbaum dazu führt, daß immer ein eigener Zweig für diese Knoten generiert wird. Abbildung 14.4 zeigt, daß diese Informationen dann aber immer in den parallelen Zweigen Verwendung finden, so daß zunächst der Schemaname und der Schematyp jeweils in den im Syntaxbaum oberhalb liegenden Knoten kopiert werden müssen. Die Vorgehensweise dafür ist bereits aus dem vorhergehenden Abschnitt 14.3 bekannt. Die notwendigen Spezifikationen Eli betreffend und die speziell hier angewandte Vorgehensweise können im Anhang E.3 auf der Seite 113 nachgelesen werden.

Im weiteren wird nun in die beiden Mappingrichtungen '*from*' und '*to*' unterschieden, da sich unterschiedliche Strategien bei den notwendigen Berechnungen ergeben haben.

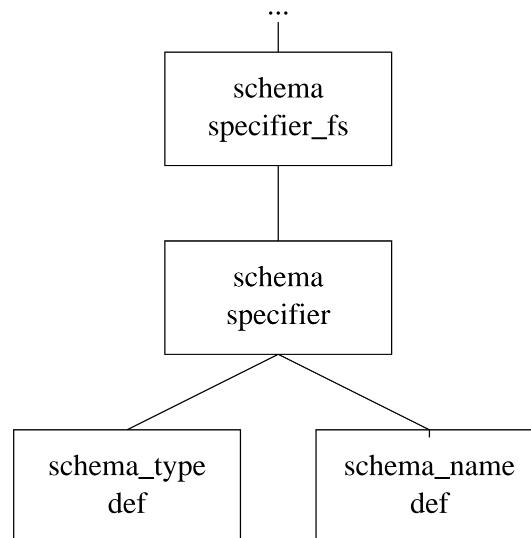

Abbildung 14.4: Ausschnitt aus dem Syntaxbaum.

14.4.1 Mapping-From

Beim Mapping werden in dieser Sprache grundsätzlich drei Objekte vom gleichen Typ miteinander verbunden. Den Anfang bildet dabei das Verbinden von Schemata, innerhalb dieser folgen Pointer, Listen und Klassen und innerhalb von Klassen wiederum noch Attribute. In allen Fällen werden die Namen und Typen der drei beteiligten Schemata benötigt. Im Fall der Attributverbindungen darüber hinaus auch die Namen der zugehörigen Klassen. Wenn durch den Scanner das Symbol `mapping_schema_from` erkannt wird, so ist an den Verbindungen neben dem föderierten und dem Komponentenschema noch ein Exportschema beteiligt. Diese Namen müssen zunächst für die weitere Verwendung in den gemeinsamen Knoten `mapping_schema_from` kopiert werden – die auch hier vorhande Problematik der parallelen Zweige läßt sich aus Abbildung 14.5 erkennen. Hierzu werden diesmal für *jeden* Schematyp jeweils ein Attribut für den Typ und den Namen des Schemas angelegt.

Im Syntaxbaum (Abbildung 14.5) folgt dem Knoten `mapping_schema_from` der Knoten `mapping_type_from`. In diesen werden zunächst aus Gründen der besseren Übersicht die Schemanamen und Schematypen von vorhin kopiert. Zusätzlich dazu verfügt der Knoten `mapping_type_from` über drei Zweige, in denen jeweils die in die Schemata passenden Klassen angegeben sind. Auch diese Informationen müssen in dem Knoten `mapping_type_from` gesammelt werden, um diese später verwenden zu können. Es werden also noch drei weitere Attribute angelegt und aus den drei Zweigen jeweils das richtige Attribut kopiert. Die Grafik in Abbildung 14.6 zeigt dieses *Zusammensammeln* der einzelnen Attribute, die für die spätere Ausgabe wichtig sind. Aus Gründen der Übersicht wurde in der Grafik auf einige Knoten `schema_type_def` und `schema_name_def` verzichtet. Auf die Knoten zwischen `mapping_schema_from` und `schema_type_def` bzw. `schema_name_def` wurde ebenfalls verzichtet. Es sind also nur die Endstellen des gesamten Kopiervorganges aufgeführt, um die zugrundeliegende Problematik zu verdeutlichen. Aus der Abbildung 14.5 läßt sich jedoch die vollständige Baumstruktur schnell ableiten. Ebenfalls aus Gründen der Übersicht wurden nur die Pfeile für das Kopieren der FS-Knoten eingezeichnet. Für die übrigen aufgeführten Attribute gelten natürlich eigene analoge Pfeile.

Die bislang aufgeführten Knoten des Syntaxbaumes führen allerdings zu keinerlei Ausga-

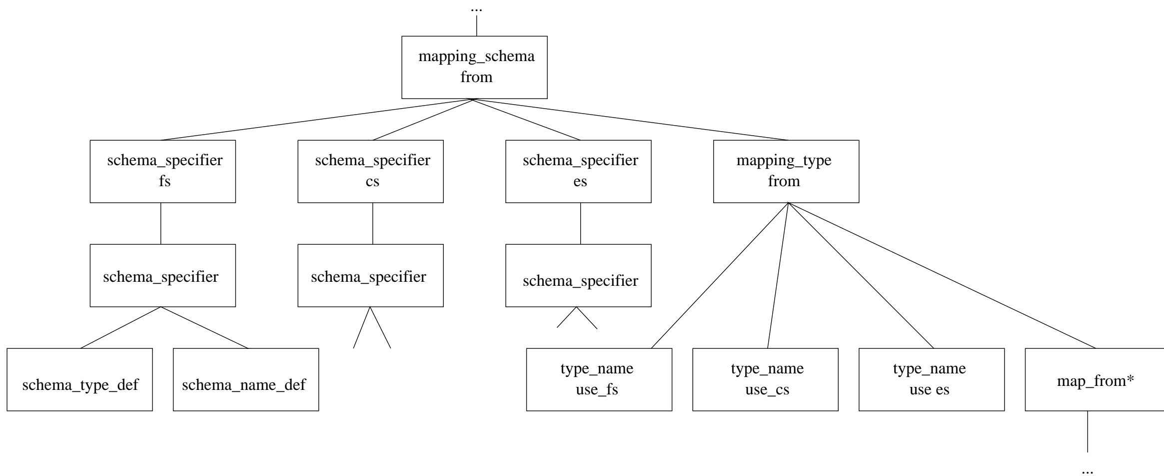

Abbildung 14.5: Ausschnitt aus dem Syntaxbaum.

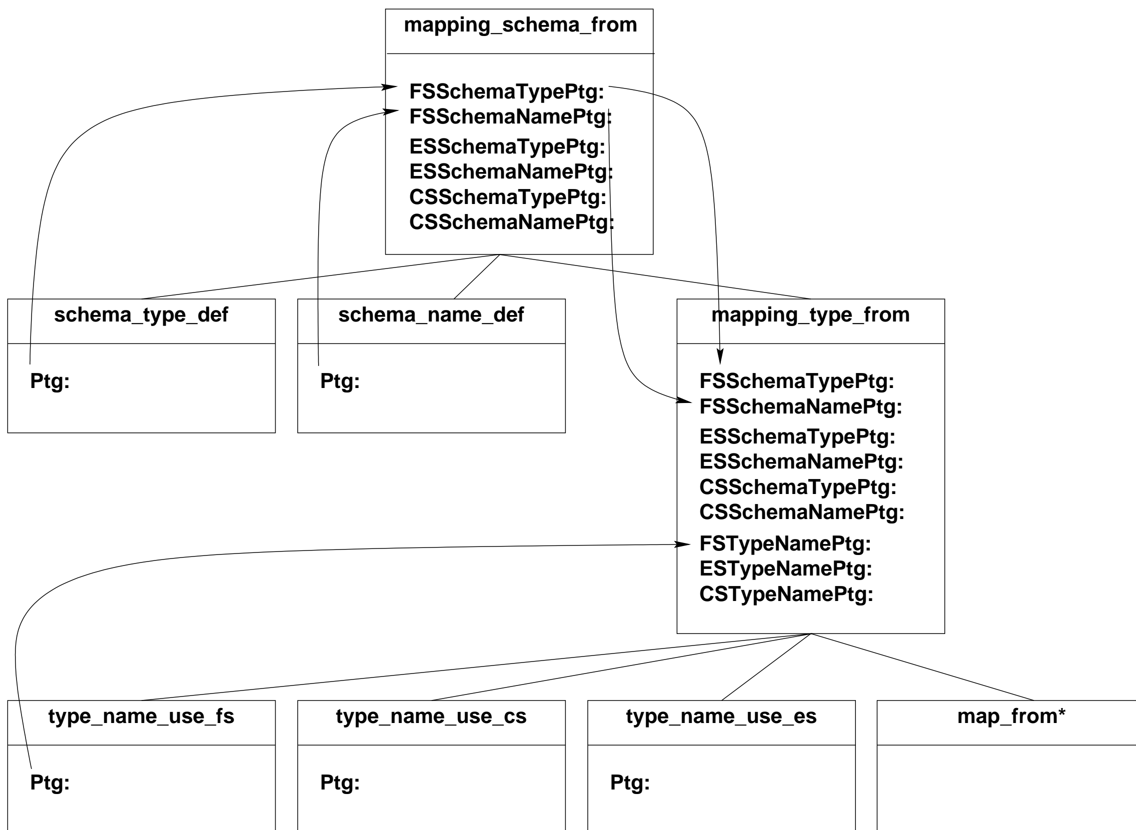

Abbildung 14.6: Kopieren der notwendigen PTG-Attribute in den Knoten `mapping_type_from`.

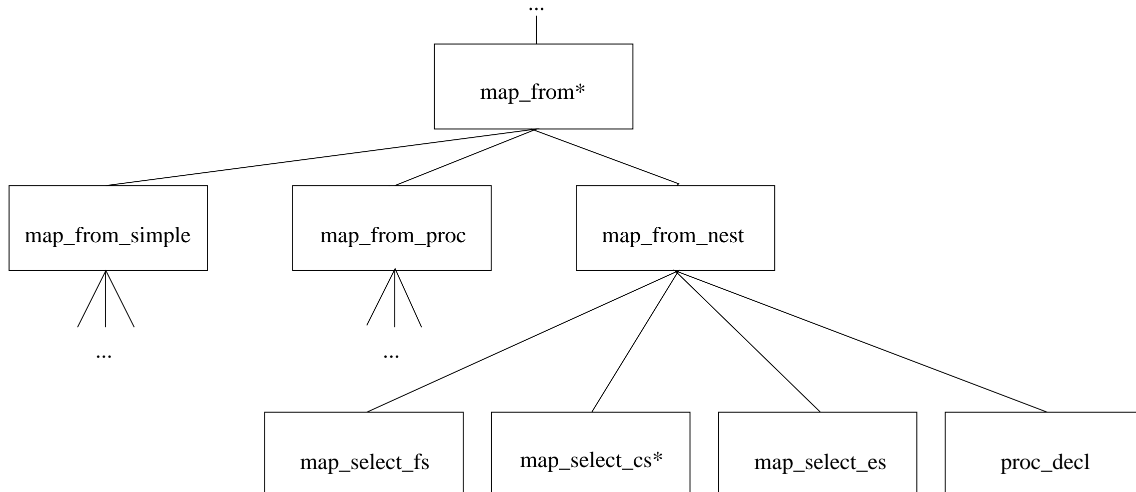

Abbildung 14.7: Ausschnitt aus dem Syntaxbaum für den Bereich 'Mapping-From'.

be. Sie beinhalten lediglich dazu notwendige Informationen. Die Knoten `map_from_simple`, `map_from_proc` und `map_from_nest` dagegen führen später zu Ausgaben. Damit bei den dazu später notwendigen Funktionsaufrufen die Parameterübergabe so einfach wie möglich ist, werden in diese Knoten nun die gesammelten Daten über Schemanamen, Schematypen und Klassennamen propagiert.

Für die Knoten `map_from_simple` und `map_from_proc` sind dies alle durchzuführenden Berechnungen. Die gesamten Spezifikationen für diese Kopiervorgänge, sowie eine ausführliche Erläuterung der Spezifikationskonstrukte in Zusammenhang mit einer Auffrischung der Vorgehensweise ist im Anhang E.3.1 auf den Seiten 114 ff. zu finden.

14.4.2 Map-From-Nest

Eine Ausnahme bei den Berechnungen bildet hierbei der Knoten `map_from_nest`. Die Grammatik erlaubt hierbei die Angabe mehrerer Attribute im Komponentenschema. In diesem Fall führt also jedes dieser Attribute zu einer Ausgabe. Im Gegensatz dazu waren bei den Knoten `map_from_simple` und `map_from_proc` nur eine Ausgabe für jedes Attribut notwendig. Die Funktionsaufrufe für den Ausgabetext werden sich in diesem Fall also nicht an dem Knoten `map_from_nest` orientieren, da dieser nur zu einer Ausgabe führen würde, sondern an der Menge von `map_select_multi_cs`-Knoten. In all diesen Knoten werden also in diesem Fall auch die Informationen über Schemanamen, Schematypen und Klassennamen benötigt. Diese Daten werden demnach auch in diese Knoten kopiert, um im späteren Verlauf leichter auf die Daten zugreifen zu können. Für die Ausgaben, die von diesen Knoten veranlaßt werden, wird allerdings zusätzlich der Name des Attributes im Exportschema, also der Inhalt des Knotens `map_select_es` benötigt. Dieser Knoten liegt nun aber wiederum in einem parallelen Zweig des Syntaxbaumes. Aus diesem Grunde wird auch dieses notwendige Datum zunächst in den übergeordneten Knoten `map_from_nest` kopiert. Die so in diesem Knoten gesammelten Attribute werden dann, wie schon des öfteren praktiziert, in jeden vorhanden Knoten `map_select_multi_cs` kopiert, weil sie dort später benötigt werden, bzw. weil deren Zugriff dadurch erheblich vereinfacht wird. Die Abbildung 14.7 verdeutlicht die hier vorliegenden parallelen Zweige.

```
MAP_SCHEMA FS.KrKh TO CS.Angio VIA IS.Angio
  MAP_TYPE SchwesterOben TO SchwesterUnten VIA SchwesterMitte
    MAP VornameOben,  NachNameOben TO NameUnten
      VIA VornameMitte, NachNameMitte
      PROC concat
    END_MAP
  END_MAP_TYPE
END_MAP_SCHEMA
```

Abbildung 14.8: Ein kleines Beispiel für ein Nesting.

Mit Hilfe dieser Knotenberechnungen stehen nun also in den Knoten, die später zu einem Aufruf einer PTG-Ausgabefunktion führen, die notwendigen Übergabeparameter in Form von PTG-Knoten zur Verfügung.

Die ausführlichen Spezifikationen, die von Eli verwendet werden, und deren Erläuterungen finden sich im Anhang E.3.2 ab der Seite 115.

14.4.3 Mapping-To

Wenn im Quelltext das Mapping-To Konstrukt verwendet wird, so werden für die dazugehörenden Ausgaben in der Zielsprache in weiten Teilen dieselben Daten benötigt wie im vorhergehenden Mapping-From. Anstelle eines Exportschemas ist in diesem Falle ein Importschema beteiligt. Die sich dadaurch ergebenden Änderungen in den Spezifikationen äußern sich lediglich durch die Verwendung eines anderen Symbolnamens. Um also die für die Ausgabe relevanten Daten in den entsprechenden Knoten des Syntaxbaumes zur Verfügung zu stellen, gilt dieselbe Vorgehensweise wie beim Mapping-From. Zunächst werden die Schemanamen und Schematypen der drei beteiligten Schemata in den gemeinsamen Knoten `mapping_schema_to` kopiert. Diese Angaben werden in den darunter liegenden Knoten `mapping_type_to` kopiert und in diesem zusätzlich die Daten über die beteiligten Klassennamen gesammelt – analog zu dem `mapping_type_from`. Für die nun im Syntaxbaum folgenden Knoten `map_to_simple` und `map_to_proc`, die später zu einer Ausgabe führen, werden nun diese gesammelten Daten in die beiden Knoten propagiert. Für den Fall, daß ein `map_to_nest`-Symbol erkannt wird, kommt es auch hier zu einer Ausnahme, die weitere Berechnungen erforderlich macht.

14.4.4 Map-To-Nest

In diesem Abschnitt wird etwas näher auf die Verwendung von Eli eingegangen, weil sich dadurch gegenüber der ursprünglichen Version der BNF eine Änderung in der Grammatik ergibt.

Die Problematik beim Map-To-Nest läßt sich am besten durch das kleine Beispiel aus Abbildung 14.8 verdeutlichen.

In diesem Beispiel wird nur ein Ausschnitt aus einem größeren Quellprogramm dargestellt. Es seien ein föderiertes Schema “KrKh”, ein Komponentenschema “Angio” und ein Importschema “Angio” an dem Mapping beteiligt. Zur besseren bildlichen Vorstellung sei angenommen, daß man sich das föderierte Schema oben vorstellt, darunter das Importschema in der Mitte und schließlich das

```
map_to_nest : 'MAP' map_select_fs ',' map_select_fs
                                     (',' map_select_fs)*
                   'TO'  map_select_cs
                   'VIA' map_select_is ','
                         map_select_is (',' map_select_is)*
                   proc_decl
              'END_MAP' .
```

Abbildung 14.9: Ausschnitt aus der Grammtik.

Komponentenschema am unteren Ende. Daraus seien dann auch die beteiligten Klassen des Mappings benannt: SchwesterOben, SchwesterMitte und SchwesterUnten. Die zwei, von dem Mapping betroffenen Attribute seien ebenfalls nach dieser bildlichen Anordnung benannt: VornameOben, VornameMitte, NachnameOben und NachnameMitte. Diese Attribute sollen zusammen in das Attribut NameUnten *gemappt* werden. Die dazu notwendige Konvertierungsmethode sei mit "Concat" benannt.

Aus dem Beispiel wird deutlich, daß die Reihenfolge der Attribute beim Mapping eine Rolle spielt: es wird von VornameOben über VornameMitte nach NameUnten und von NachnameOben über NachnameMitte nach NameUnten gemappt. Der Compiler muß für die Zielsprache für jedes Attribut des föderierten Schemas, also der ersten Liste, eine Ausgabe generieren, die dieses Attribut mit dem jeweils richtigen Attribut der zweiten Liste verbindet. Dazu gehört für jedes Element der zweiten Liste eine Ausgabe, die dieses mit dem einen Attribut des Komponentenschemas verbindet. Die Generierung einer Ausgabe für jedes vorkommende Listenelement ist keine Besonderheit, denn derartige Ausgaben wurden schon früher benötigt. Die Besonderheit bildet die zuvor genannte Ausgabe. Es muß eine Liste abgearbeitet werden, also z.B. das k-te Element und es wird das dazu passende Element der zweiten Liste, also dort auch das k-te Element benötigt. Im Rahmen einer gewöhnlichen Programmierung von Listen in einer beliebigen Programmiersprache würde eine solche Aufgabenstellung keine Problematik aufwerfen. Bei Berechnungen in einem Syntaxbaum unter Verwendung von Eli zunächst schon. Allerdings helfen auch hier Bibliotheken von Eli weiter und das Problem kann auch in diesem Falle durch Beschreibung gelöst werden. Eli stellt im Modul `LidoList` die Verwaltung einer eli-eigenen Listenform zur Verfügung, welche scheinbar sehr geeignet für die Lösung derartiger Probleme ist. Sie ermöglicht es einem Symbol in der Grammatik, respektive einem Knoten des Syntaxbaumes, eine Liste in Form eines neuen Attributes anzuhängen. Die Listenelemente werden dabei aus diesem Knoten nachfolgenden Blättern entnommen. Diese Liste kann dann zu einem späteren Zeitpunkt der Reihe nach wieder "verbraucht" werden, mit anderen Worten bei jedem Aufruf eines Listenelementes wird das erste Listenelement zurückgeliefert und anschließend aus der Liste gelöscht.

Für den in Abbildung 14.9 gezeigten Ausschnitt aus der Grammatik würde dieses Listenmodul zunächst die Überlegung zulassen, zwei solcher Listen für das Symbol `map_to_nest` zu berechnen und diese dann beide parallel abzuarbeiten. Die Listeninhalte wären dann jeweils sämtliche `map_select_fs` und `map_select_is` gewesen. Dabei stellt sich allerdings heraus, daß bei der Verwendung dieser `LidoList` der Attributname der Liste eigenständig aus dem Symbolnamen und den Modulfunktionen zusammengesetzt wird, so daß in jedem Symbol nur jeweils eine einzige Liste definiert werden kann. Auch ist die Abarbeitung der Liste in demselben Symbol nicht möglich. Denn innerhalb desselben Symboles kann nicht spezifiziert werden, daß eine Liste erstellt werden soll und daß diese abgebaut werden soll. Aus diesem Grunde ist eine andere Vorgehensweise notwendig. In

```
map_to_nest    : 'MAP' fs_part
                       'TO' map_select_cs
                       'VIA' is_part
                       proc_decl
                 'END_MAP' .
   fs_part: map_select_multi_fs ',' map_select_multi_fs
            (',' map_select_multi_fs)* .
   is_part: map_select_multi_is ',' map_select_multi_is
            (',' map_select_multi_is)* .
```

Abbildung 14.10: Abgeänderte Grammatik für das Symbol map_to_nest.

die Grammatik der Sprache werden in dem Symbol map_to_nest zwei weitere Symbole eingeführt (siehe Abbildung 14.10), die jeweils die entsprechenden Listen spezifizieren. Daraufhin ist es möglich in einem der beiden Symbole die gewünschte Liste zu generieren, diese in das andere Symbol zu kopieren und dort als *Abbau*-Liste zu spezifizieren. Die einzelnen Elemente der anderen Liste werden dann, wie aus anderen Berechnungen bekannt, direkt mittels Symbolberechnungen durchlaufen.

Da für jedes Element der is_part-Liste zwei Ausgaben notwendig sind, wird diese Liste mittels Symbolberechnung durchlaufen und für das Symbol fs_part die notwendige abbaubare Liste konstruiert.

Das Modul LidoList wartet noch mit einer weiteren, schlecht dokumentierten Besonderheit auf. Der Attributname in den Knoten, dessen Inhalt in die Liste aufgenommen werden soll, ist ebenfalls durch das Modul vorgegeben. Er setzt sich aus dem ElementTyp und dem Wort *Elem* zusammen. In jedem der für die Liste benötigten Knoten wurde in früheren Spezifikationen bereits ein Attribut namens Ptg generiert, welches den ausgabefähigen Code beinhaltet. Der Typ dieses Attributes ist PTGNode. Für alle Knoten map_select_multi_fs muß also der Inhalt des Ptg-Attributes in ein neues Attribut mit dem Namen PTGNodeElem kopiert werden. Der Vorgang des Kopierens ist mitlerweile ausreichend diskutiert.

Nun kann im Knoten fs_part die gewünschte LidoList konstruiert werden und anschließend für den Abbau in den Knoten is_part kopiert werden. Die für Eli dazu notwendigen *neuen* Spezifikationskonstrukte nebst ihrer Vor- und Nachteile und die ausführliche Erläuterung der Verwendung der LidoList und der dazugehörenden Module kann im Anhang E.3.2 ab den Seiten 115 nachgelesen werden.

Die restlichen Berechnungen, die noch für das Map-To-Nest notwendig sind, ähneln denen des Map-From-Nest und können ebenfalls im Anhang E.3.3 ab der Seite 116 nachgelesen werden.

Die Beschreibung der Spezifikationsdateien findet sich in Angang E.4 auf Seite 118.

14.5 Die Ausgabe

Bislang wurde in diesem Kapitel beschrieben, welche Berechnungen notwendig sind, um die für die Ausgabe notwendigen Daten in den entsprechenden Knoten zu sammeln und wie für die entsprechenden Blätter des Syntaxbaumes Ptg-Attribute generiert wurden, dessen Werte den String des Knotens im Eingabetext repräsentieren. In Kapitel 13 wurde bereits beschrieben wie mit Hilfe von PTG-Mustern die Struktur der Ausgabe geformt werden kann. Diese Muster wurden bislang noch

nicht verwendet. Dieser Abschnitt spezifiziert nun die verschiedenen Aufrufe dieser Muster mit den dazugehörenden Parametern. Eine starke Verschachtelung dieser Aufrufe generiert eine Baumstruktur von PTGNodes, welche den Ausgabetext repräsentiert. Letztlich wird dann dieser Baum an die Funktion `PTGOut` übergeben, welche die Datenstruktur linearisiert und ausgibt. Die Funktionsaufrufe sind auch in diesem Fall an Symbolen der Grammatik verankert. Es werden demnach für alle betroffenen Knoten im Syntaxbaum ebenfalls Attribute mit dem Namen Ptg berechnet, welche zur Wurzel des Baumes hin immer komplexer werden, da sie aus darunter liegenden PTGNodes zusammengesetzt werden. Bei der Spezifikation dieser Symbolberechnungen wird bei der Wurzel des Baumes begonnen und diese Schritt für Schritt bis hin zu den Knoten vor den Blättern des Baumes verfeinert.

Eine umfassende Erläuterung der einzelnen, notwendigen Schritte beim Durchlauf durch den Syntaxbaum, der verwendeten, teilweise *neuen* Spezifikationskonstrukte und der daraus resultierenden Spezifikationen bezüglich Eli finden sich im Anhang E.5 auf den Seiten 118-123. Die für Eli notwendigen Spezifikationsdateien sind schließlich im Anhang E.6 auf Seite 123 erläutert.

Die Spezifikation des gewünschten Compilers ist somit abgeschlossen. Eine lauffähige Version dieses Compilers kann nun, wie in Kapitel 8.5 beschrieben, generiert werden und in Verbindung mit einer geeigneten Eingabe gestartet werden.

Kapitel 15

Das Laufzeitsystem in O_2

In diesem Kapitel werden die Komponenten erläutert, die notwendig sind, um später die Ausgaben des Compilers nutzbar zu machen. In der Ausgabe des Compilers werden nur Methoden definiert, die dazu dienen einen Föderierungsgraphen aufzubauen. Diese bedienen sich natürlich einer ganzen Reihe von weiteren Strukturen (im wesentlichen *C++*-Klassen), die ebenfalls implementiert werden müssen, bevor sie in der O_2-Datenbank nutzbar gemacht werden können.

15.1 Die Weiterverarbeitung der Compilerausgabe

Der vom Compiler generierte Code muß zunächst in eine Datei ausgegeben werden. Ein Blick in diesen *C++*-Quellcode zeigt, daß dort die Methoden `BuildGraph` und `BuildSimpleTypes` definiert werden. Wie bei der *C++*-Programmierung üblich wird in diesem Code eine Deklarationsdatei eingebunden, die der Compiler fest mit dem Namen `build.H` angibt. Diese Datei wird vom Compiler nicht generiert, denn ihr Inhalt ist vom variablen Ergebnis des Compilers unabhängig. Die Deklarationsdatei braucht demnach nur die beiden erwähnten Methoden deklarieren und sieht wie folgt aus:

```
// build.H
#include "graph.H"

// Funktionsprototypen
void BuildGraph( PCGraph pg );
void BuildSimpleTypes( PCSchema ps );
```

Diese Datei muß also vorhanden sein, damit es möglich ist einen Graphen in die O_2-Datenbank zu schreiben.

15.2 Die Portierung der Graphklassen

Die Grundlage für die Graphstruktur wurde in [Pro97] gelegt. Dort wurden auch die notwendigen Graphklassen definiert und implementiert. Da das grundlegende Konzept des Graphen von dort übernommen wurde, können auch die Graphklassen wiederverwendet werden. In [Pro97] wurde der Föderierungsgraph im Hauptspeicher des Rechners gehalten. Eine der großen Erweiterungen ist den Gra-

phen nun persistent in der O_2-Datenbank zu speichern, so daß in ihm enthaltene Föderierungsinformationen nicht mehr verloren gehen, wenn das System abgeschaltet wird.

Ohne weiteres können die *alten* Klassen allerdings nicht verwendet werden. Im Folgenden wird erläutert welche Änderungen vorgenommen werden mußten und wie die Klassen in die Datenbank importiert werden können.

15.2.1 Vorbereiten der Graphklassen

Eine grundlegende Eigenschaft der O_2-Datenbank besteht darin, daß sie nicht mit *fremden* Template-Klassen zusammenarbeitet. Obwohl in der Ursprungsversion der Graphklassen ausgiebig von diesem programmiertechnischen Mittel Gebrauch gemacht wurde, konnte für die Portierung auf eine teilweise von *C++*-Templates bereinigte Version der Graphklassen zurückgegriffen werden. [1]

Die Datenbank O_2 stellt mittels des sogenannten *C++*-Bindings [O296b] die Möglichkeit zur Verfügung normale *C++*-Klassen bzw. deren Objekte in der Datenbank zu verwenden. Dazu müssen die Klassen beim Instanziieren eines Objektes im *C++*-Quellcode mit einem `d_Ref<Klasse>` versehen werden. Dies führt dazu, daß keine *C++*-eigenen Zeiger verwendet werden, sondern Zeiger, die auf entsprechende Objekte in der Datenbank deuten. In [Pro97] wurde programmiertechnisch konsequent mit selbstdefinierten Typen gearbeitet. Dies bedeutet u.a., daß für alle Zeiger auf Objekte in der gemeinsamen Deklarationsdatei ein eigener Typ definiert wurde. Somit wurde die Portierung der Klassen sehr vereinfacht, denn es hätten in der Deklarationsdatei nur diese selbstdefinierten Typen um die erwähnten `d_Refs` erweitert werden müssen. Ein Beispiel:

Vorher: `typedef CConvert PCConvert;`
Nachher: `typedef d_Ref<CConvert> PCConvert;`

Leider stellte sich dabei heraus, daß O_2 innerhalb von einem `typedef` nicht mit `d_Ref` arbeiten kann, sodaß auf die *C++*-Compilermöglichkeit des `define` zurückgegriffen werden mußte. Dies bedeutet, daß alle `typedef` durch einen entsprechenden `define`-Ausdruck ersetzt werden mußten. Ein Beispiel:

Vorher: `typedef CConvert PCConvert;`
Nachher: `#define PCConvert d_Ref<CConvert>`

Der nächste Schritt bei der Vorbereitung der Klassen ist das Bereinigen der Klassen von den verwendeten STL-Templates [Jos96]. Aus diesen Templates wurden *Strings* und *Listen* verwendet. Die Datenbank O_2 bietet im Rahmen des *C++*-Bindings ein eigenes String- und Listentemplate an. Für die Verwendung des O_2-eigenen `d_String` müssen also im gesamten Quelltext der Graphklassen und in der dazugehörenden Deklarationsdatei `String` durch `d_String` ersetzt werden.

Bei den Listen ist die Portierung nicht ganz so einfach, denn Navigationen auf STL-Listen und O_2-Listen sind sowohl syntaktisch wie auch in ihrer Vorgehensweise stark unterschiedlich. Für die Portierung wurden also zunächst im gesamten Quellcode der Graphklassen jede Verwendung von `List` durch ein `d_List` ersetzt. Danach mußten alle Methoden, die auf Listen zugreifen, neu implementiert werden. Dies betraf im wesentlichen Navigationen innerhalb von Listen.

In [Pro97] wurde für alle benutzten Graphklassen eine einheitliche Deklarationsdatei mit dem Namen `graph.H` verwendet, um die Verwendung der insgesamt 13 Klassen einfacher zu gestalten. Dieses

[1] An dieser Stelle sei Peter Ziesche, der auch an der Projektgruppe teilgenommen hat, gedankt. Er hatte sich schon vor dem eigentlichen Beginn der Diplomarbeit bereiterklärt die Graphklassen, die von ihm implemeniert wurden, von seinen selbstgeschriebenen Templateklassen zu befreien.

Prinzip wurde auch bei der Portierung beibehalten, sodaß auch diese Datei entsprechend abgeändert werden mußte.

Die so abgeänderten, bzw. reimplementierten Klassen stehen nun für die Benutzung in Zusammenhang mit O_2 zur Verfügung.

15.2.2 Import der Klassen in O_2

Alle Klassen, die später einmal in der Datenbank benutzt werden, müssen zuvor in diese importiert werden. Auch dies ist ein Bestandteil des *C++*-Bindings von O_2. Nachdem die O_2-Datenbank eingerichtet ist, muß in ihr ein *Schema* angelegt werden. Innerhalb von *Schemata* können später die eigentlichen Datenbanken angelegt werden. Das Schema muß allerdings von Hand angelegt werden und kann nicht durch das Laufzeitsystem generiert werden. Die Verwendung des *C++*-Bindings bringt die Benutzung des O_2-eigenen *Makefilegenerators* [O297] mit sich, denn an der Compilierung von eigenen *C++*-Quellcodes ist ab sofort nicht nur ein gewöhnlicher *C++*-Compiler beteiligt, sondern auch immer Komponenten von O_2. Mittels einer Konfigurationsdatei kann das O_2-Werkzeug ein Makefile generieren, welches benutzt werden muß, um eigene Anwendungen zu übersetzen oder auch nur die verwendeten Klassen in die Datenbank zu importieren. In diesem Konfigurationsfile werden u.a. für den Import die Deklarationsdatei der verwendeten Graphklassen, die Namen der Graphklassen selbst nebst ihrer Quellcodedateien und vor allem auch alle Typen von verwendeten Listen angegeben. Eine Konfigurationsdatei für eine Anwendung, die den Föderierungsgraphen verwendet, findet sich als Beispiel im Anhang F auf der Seite 125. Es handelt sich dabei um die Konfigurationsdatei für den Föderierungskern des Testsystems (siehe 17, Seite 79).

Wurde auf diese Weise eine Makefile generiert, so sollte der Import der Klassen durch den Aufruf von '`make import`' möglich sein.

15.3 Benutzung der Graphklassen

Nachdem die Graphklassen in die Datenbank importiert wurden, können diese in eigenen Anwendungen verwendet werden. Dazu müssen in der eigenen Anwendung die folgenden Deklarationsdateien eingebunden werden:

graph.H - die Deklarationsdatei der Graphklassen.

o2lib_CC.hxx - O_2-interne Deklarationen.

o2template_CC.hxx - O_2-interne Deklarationen.

o2util_CC.hxx - O_2-interne Deklarationen.

Um das eigene Programm zu übersetzen müssen die selbstgeschriebenen Quellcodes zusätzlich in die Konfigurationsdatei für den Makefilegenerator eingetragen werden. Die genauen Beschreibungen zur Verwendung der Datenbank sind den entsprechenden Handbüchern zu entnehmen.

Es wurde eine Anwendung implementiert, die eine Datenbank anlegt und in diese den spezifizierten Föderierungsgraph einfügt. Diese Anwendung bindet zu diesem Zwecke die Deklarationsdatei `build.H` ein und bedient sich daher für den Aufbau des Graphen der Compilerausgabe. Die Anwendung wird mit `buildgraph` aufgerufen.

Kapitel 16

Zusammenfassung

In diesem Kapitel wird ein Rückblick über die während des Entwicklungsprozesses gewonnenen Erfahrungen gegeben und ein Ausblick auf eventuelle Erweiterungsmöglichkeiten zu dem Compiler diskutiert.

16.1 Rückblick

Insgesamt konnte ein Prototyp des spezifizierten Compilers realisiert werden, der den grundlegenden Ansprüchen nachkommt und der aus einer vorgegebenen, korrekten Spezifikation des Föderierungsgraphen in der entwickelten Sprache eine korrekte Ausgabe generiert, die zum Aufbau des Föderierungsgraphen in der O_2-Datenbank genutzt werden kann. Dadurch, daß der generierte Code, sogar unabhängig von den O_2-spezifischen Programmierelementen geblieben ist, kann der gewonnene *C++*-Code sogar ohne Datenbank oder in Verbindung mit anderen objektorientierten Datenbanken nutzbar gemacht werden.

16.1.1 Das Compilerbauwerkzeug ELI

Einer der Hauptgründe für die Verwendung von Eli war, daß das Werkzeug den Entwurf eines gesamten Compiler unterstützt und daß es versprach auch anwendbar zu sein, wenn der Benutzer über die Feinheiten des Compilerbaus kein umfassendes Hintergrundwissen verfügt. In diesem Sinne hat sich Eli als wirklich kompetentes Werkzeug herausgestellt, da es über eine sehr große Ansammlung an Bibliotheken für den Compilerbau verfügt. Dadurch, daß der gesamte Bau des Compilers durch Eli-Spezifikationen und nicht durch eigene Implementierungen möglich war, stellt sich das Ergebnis als sehr kompakt dar. Für die gesamte Spezifikation des Compilers sind nur rund 1520 Zeilen notwendig gewesen. Eine eigenständige Implementierung eines solchen Compilers hätte dies um ein Vielfaches überragt.

Der größte Nachteil, der bei Eli zum Vorschein kommt, ist der hohe Zeitaufwand für die Einarbeitung in dieses Werkzeug. So wenig Wissen über Compilerbau notwendig ist, um so mehr Wissen ist über Eli erforderlich. Als besonders gravierend hat sich dabei die undurchsichtige und nicht besonders verständliche Anleitung von Eli herausgestellt. Selbst das darin enthaltene *Running Example* hat sich nicht als sonderlich nützlich erwiesen. Ein Einstieg an einer bestimmten Stelle in dieses Beispiel ist fast nicht möglich, da entsprechende Kenntnisse über die Vorgeschichte notwendig sind, um das

Beispiel zu verstehen. Mehrere kleine Beispiele für die entsprechenden Module und ihre Verwendung wären da sicherlich hilfreicher gewesen. Die Anleitung selbst bleibt sehr Eli-spezifisch und ist wenig erklärend. Für das Verständnis und die spätere Fähigkeit einer eigenen Spezifikation war es erforderlich mehrere große Compiler-Spezifikationen für Eli, die frei zugänglich sind, zu untersuchen und mehrere Veröffentlichungen über dieses Thema zu studieren. Zum Lösen bestimmter Probleme verging dabei sehr viel Zeit. Eine sehr große Hilfe fand sich später in der Support-Mailingliste von Eli. Hier konnte man auf meist schnelle Hilfe bei Problemen hoffen, wenn auch des öfteren von dort wieder auf entsprechende Abschnitte in der Anleitung verwiesen wurde.

Zusammengefaßt läßt sich feststellen, daß eine Wiederverwendung von Eli nun sicherlich leichter fallen würde, da das System vertrauter ist und die Informationsquellen bekannt sind. Es ist allerdings genau abzuwägen, ob ein so mächtiges Werkzeug eingesetzt werden muß und der Einarbeitungsaufwand gerechtfertig ist. Um einen einfachen Compiler ohne Hintergrundwissen über Compilerbau zu generieren ist Eli durchaus zu empfehlen. Darüber hinaus ist auch für Eli ein größeres Wissen über den Compilerbau notwendig.

16.1.2 Die Datenbank O_2

Die Verwendung der Datenbank erwies sich als sehr zeitaufwendig. Der Einarbeitungsaufwand für diese Datenbank ist groß. Ist die Vorgehensweise für das *C++*-Binding ersteinmal ausreichend bekannt und sind die ersten Programmierschritte gemacht, so ist es jedoch relativ leicht möglich *C++*-Objekte in einer Datenbank abzulegen und persistent zu machen. Der Zugriff auf diese Klassen und vor allem auch auf Methoden in diesen Klassen ist danach sogar halbwegs transparent und ohne Aufwand möglich. Es gibt jedoch so einige Einschränkungen, die man bei der Benutzung des *C++*-Bindings in Kauf nehmen muß. Es ist nicht möglich den volle Funktionsumfang von *C++* bzw. alle Möglichkeiten der objektorientierten Programmierung unter *C++* in die Datenbank zu überführen. So mußten beispielsweise große Teile der Graphklassen umprogrammiert bzw. auf andere Templateklassen umgestellt werden. Auch war es z.B. nicht möglich ein Objekt als sogenannte *persistent Root* zu definieren, obwohl dies in der Anleitung ausdrücklich als machbar erwähnt wurde. Es war letztlich nur möglich durch eine einelementige Liste dieses Problem zu umgehen. Nicht zuletzt aus diesem Grunde erscheint das Produkt als noch unausgereigt.

Der größte Nachteil bei der Programmierung in Verbindung mit O_2 sind die absolut aussagelosen Fehlermeldungen, sowohl bei der Übersetzung von eigenen Programmen wie auch bei auftretenden Laufzeitfehlern. Sie stellen den Entwickler während der Programmierung immer wieder vor scheinbar unlösbare Probleme. Auch der Einsatz von Debuggern ist hier nicht von großer Hilfe, da die Fehler ausschließlich in O_2-internen Methoden auftreten und in keinster Weise auf die Art des Fehlers hindeuten. Die Fehlersuche ist somit ungemein Zeitaufwendig und erscheint in vielen Fällen als absurd, weil das Fehlergebiet nicht eingegrenzt werden kann. Während der Implementierung des Testsystems wurde aus diesem Grunde mehrfach daran gedacht die Portierung aufzugeben.

Die Möglichkeit einfache *C++*-Konstrukte leicht in eine Datenbank zu transferieren und somit persitent zu machen ist sicherlich ein großer Vorteil. Die Nachteile bei der Programmierung überwiegen allerdings.

16.2 Ausblick

Die möglichen Erweiterungen, die für den Compiler diskutiert werden können, beziehen sich zunächst im Wesentlichen auf den erhöhten Bedienkomfort im Umgang mit dem Compiler. Es werden nur

solche Erweiterungen aufgeführt, die keiner Änderung an der Spezifikationssprache bedürfen. Im folgenden seien einige naheliegende Möglichkeiten aufgezählt:

- Die Implementierung der Bereichs- und Typanalyse. Hierdurch können mehr Fehler in einem Eingabeprogramm ausfindig gemacht werden und somit die Anwendung sicherer gemacht werden.
- Die Ausgabe des generierten Quellcodes sollte direkt in eine Datei und nicht auf den Bildschirm ausgegeben werden. Der Name der Ausgabedatei könnte dabei sinnvoller Weise als zweiter Parameter beim Compileraufruf übergeben werden. In diesem Zusammenhang sollte auch die momentan vom Benutzer anzulegende Deklarationsdatei `build.H` vom Compiler selbst, entsprechend der Namensgebung der Ausgabe generiert werden und in der Ausgabe richtig eingebunden werden.
- Implementierung einer Ausgabe einer Benutzungshilfe beim Aufruf des Compilers mit dem Parameter `--help`.

Teil III

Evaluation

Kapitel 17

Die Testumgebung

.

17.1 Überblick

Um den Schemaeditor [Ger99] und den Compiler für die Spezifikationssprache zu testen, wurde ein einfaches Föderierungssystem implementiert. Als zugrundeliegendes DBMS für die lokalen Datenbanken findet ein relationales System auf Basis von *mySQL* [Dat] Verwendung. Das System ist frei verfügbar und bietet einen Teil der *ANSI-SQL 92* Funktionalität, was für die geplante Testumgebung völlig ausreichend ist. Der Zugriff auf die Datenbank erfolgt über ein C-API (C-Application-Programming-Interface).

Der Test soll überprüfen, ob der Schemaeditor korrekte Spezifikationen erzeugt und der Compiler aus diesen Spezifikationen korrekten *C++*-Code zur Erzeugung eines FDBS generiert. Weiterhin soll die Funktionsfähigkeit der neu in den Graphen eingebrachten Funktionen Nesting/Unnesting und Wertekonvertierung getestet werden. Über das Laufzeitverhalten des Föderierungssystems können dabei evtl. Rückschlüsse auf den Schemaditor und dessen Fähigkeit, fehlerhafte Eingaben zur erkennen und abzufangen gemacht werden.

In den folgenden Abschnitten wird zuerst das zu modellierende Szenario vorgestellt und anschließend der Ablauf der Umsetzung vom Modell zum Testsystem geschildert. In Abschnitt 17.4 werden die aus der Testphase gewonnenen Ergebnisse dargestellt.

17.2 Das Szenario

Eine geeignete Testumgebung soll die Richtigkeit des generierten Graphen in O_2 zeigen. Dazu sollen möglichst alle Funktionen des Graphen (Werte- und Typkonvertierung, Nesting/Unnesting) getestet werden können.

Aus Gründen der Übersichtlichkeit und reduzierten Komplexität werden nur zwei Datenbanken föderiert und die Beispieleingaben sollen nachvollziehbar föderiert werden können.

Folgendes Szenario soll modelliert werden:

- Zwei Büchereien sollen zusammengeführt werden.

- Eingaben werden nur in einer Datenbank möglich sein.

Abbildung 17.1 zeigt die Diagramme beider Datenmodelle. Die Modelle unterscheiden sich nur in zwei Punkten. Die Felder *Standort* und *Buchnr.* in DB 1 entsprechen dem Feld *Signatur* in DB 2. Durch die Abbildung von *Standort* und *Buchnr.* in *Signatur* soll die Arbeitsweise des Nesting getestet werden. Wertekonvertierung findet bei der Umrechnung von EURO in DM für das Feld *Preis* statt.

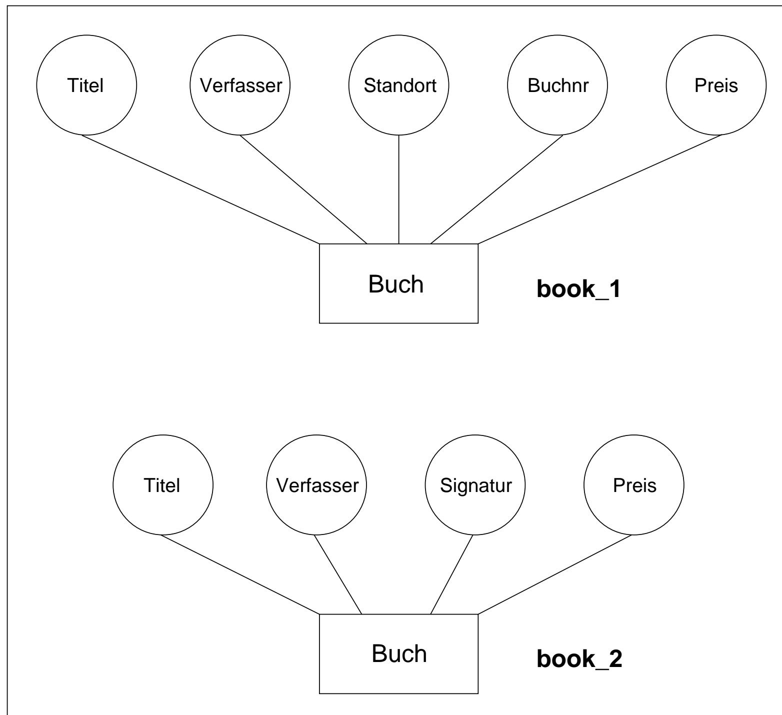

Abbildung 17.1: Entity-Diagramme für die Datenmodelle.

17.3 Ablauf

Zunächst werden die ER-Modelle für die an das Testsystem angeschlossenen Datenbanken in ein relationales Datenmodell überführt und in die entsprechenden lokalen Datenbanken eingetragen. Im zweiten Schritt werden aus den ER-Modellen die Komponentenschemata für das Föderierungssystem abgeleitet. Ausgehend von diesen Komponentenschemata wird im Schemaeditor der Föderierungsgraph spezifiziert und im Anschluß daran übernimmt der Compiler für die Spezifikationssprache die Übersetzung in *C++*-Quellcode. Aus diesem Quellcode wird der Föderierungsgraph in der O_2-Datenbank angelegt. Mit Hilfe eines speziell für dieses Szenario geschriebenen Föderierungssystems wird abschließend der Graph getestet.

17.3.1 Generierung der Datenbanktabellen

Auf Grund der Einfachheit des zu modellierenden Szenarios gestaltet sich die Umsetzung des ER-Modells für die lokalen Datenbanken in ein relationales Datenmodell sehr einfach. Abbildung 17.2 zeigt die für die lokalen Datenbanken notwendigen Tabellen.

Buch	
Titel	STRING
Verfasser	STRING
Standort	STRING
Buchnr	STRING
Preis	REAL

Buch	
Titel	STRING
Verfasser	STRING
Signatur	STRING
Preis	REAL

Abbildung 17.2: Datenbanktabellen für die lokalen Datenbanken.

Der Code zur Erzeugung der konkreten Datenbanktabellen in *mySQL* ist in Abbildung 17.3 aufgeführt.

17.3.2 Umsetzung in Komponentenschemata

Auf die explizite Überführung der lokalen Datenbankschemata in ODMG-93 konforme Komponentenschemata kann auf Grund der Einfachheit der Schemata verzichtet werden, da sich die Komponentenschemata direkt aus den Datenbanktabellen aus Abbildung 17.2 ablesen lassen.

17.3.3 Aufbau des Föderierungsgraphen

Die Eingabe des Föderierungsgraphen in den Schemaeditor [Ger99] läßt sich mit den folgenden Punkten kurz umreißen.

1. Es wird ein neues Komponentenschema mit dem Namen `book_1` angelegt.
2. Die Klasse `Buch` wird in das Schema eingetragen und mit den entsprechenden Attributen gefüllt.
3. Über die Kopierfunktion für Schemata werden Exportschema, föderiertes Schema, sowie Import- und Komponentenschema für `book_2` angelegt.
4. Im Komponentenschema `book_2` werden die Attribute `Standort` und `Buchnr` durch das Attribut `Signatur` ersetzt.
5. Die Graphstruktur wird erzeugt, indem die Schemata miteinander verbunden werden. Dazu werden jeweils ein Komponentenschema und ein Schema der Exportebene mit dem föderierten Schema verbunden. Abbildung 17.4 zeigt den Schemaeditor mit teilweise definiertem Graphen und dem Dialog zum Bearbeiten von Schemata.
6. Über den Dialog *Abhängigkeiten* werden die einzelnen Schemaelemente auf Typ- und Attributebene miteinander verbunden. Hierbei wird auch das Nesting zwischen Importschema `book_2` und Komponentenschema `book_2` ausgeführt. Für das Attribut Preis im Exportschema `book_1` wird eine Konvertierungsfunktion `EuroToDm` angegeben. In Abbildung 17.5 ist der vollständig definierte Graph im Schemaeditor zu sehen.
7. Über die Speicherfunktion wird die Spezifikation für den Föderierungsgraphen in der Datei `Books.spec` gesichert.

```
mysql << EOD

create database book_1;
create database book_2;

connect book_1;
create table BOOK
        ( Titel          varchar(80)     NOT NULL,
          Verfasser      varchar(50)     NOT NULL,
          Standort       varchar(20)     NOT NULL,
          Buchnr         varchar(10)     NOT NULL,
          Preis          real,
          PRIMARY KEY    ( Buchnr ),
          INDEX Titel    ( Titel )
        );

connect book_2;
create table BOOK
        ( Titel          varchar(80)     NOT NULL,
          Verfasser      varchar(50)     NOT NULL,
          Signatur       varchar(40)     NOT NULL,
          Preis          real,
          PRIMARY KEY    ( Signatur ),
          INDEX Titel    ( Titel )
        );

EOD
```

Abbildung 17.3: SQL-Code zur Erzeugung der Datenbanktabellen.

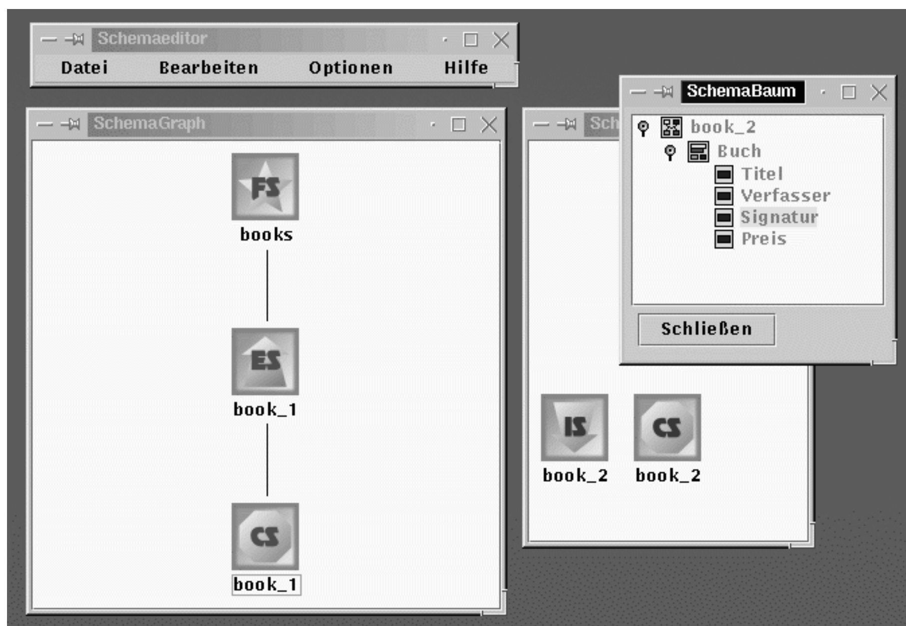

Abbildung 17.4: Schemaeditor mit partiell definiertem Graphen.

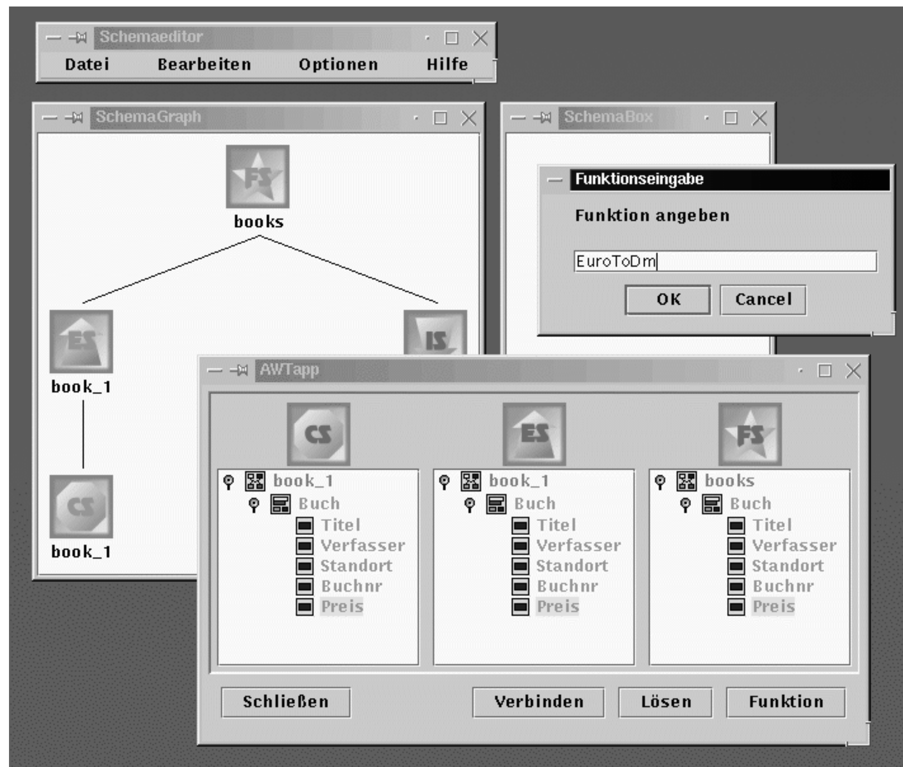

Abbildung 17.5: Schemaeditor mit vollständig definiertem Graphen.

17.3.4 Compilation der Spezifikation

Der Compiler wird mit der Spezifikationsdatei als Eingabeparameter aufgerufen. Der vom Compiler erzeugte Code wird in die Datei build.C ausgegeben:

```
compiler.exe Book.spec > build.C
```

Der so gewonnene *C++*-Code wird in ein eigenständiges Programm integriert, welches dafür sorgt, daß der Föderierungsgraph in der O_2-Datenbank angelegt wird (buildgraph). In diesem Zusammenhang müssen auch die Konvertierungsklassen `CEuroToDmConvert` und `CConcatConvert` implementiert werden. Die Realisierung der Grapherzeugung und der Konvertierungsklassen findet sich im Anhang G ab Seite 126.

17.3.5 Aufbau des Testsystems

Für den Test des Föderierungsgraphen wird ein Testsystem implementiert, welches die Komponenten aus Abbildung 17.6 enthält.

Über die lokale Datenbankapplikation (siehe Anhang G.5, Seite 136) werden Datensätze in die Datenbank `book_1` eingetragen. Gleichzeitig wird eine Stringrepräsentation des Datensatzes in eine Unix-Pipe geschrieben, die vom Föderierungskern (siehe Anhang G.4, Seite 134) ausgelesen wird. Mit Hilfe der erweiterten Graphalgorithmen (siehe Anhang G.3, Seite 131), die auf dem persistenten

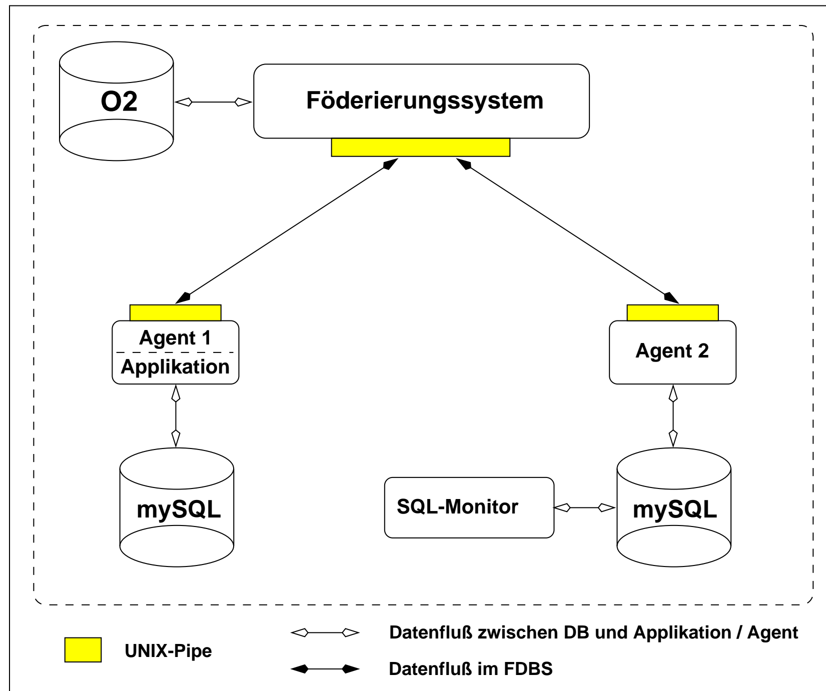

Abbildung 17.6: Die Komponenten des Testsystems.

Föderierungsgraphen in der O_2-Datenbank arbeiten, werden zu den empfangenen Datensätzen die konvertierten Datensätze für die zweite Datenbank berechnet. Anschließend werden diese vom Föderierungskern ebenfalls über eine Unix-Pipe an den Agenten der zweiten Datenbank (siehe ebenfalls Anhang G.5, Seite 136) übermittelt. Dieser fügt die erhaltenen Datensätze in die zweite Datenbank ein. Mittels eines SQL-Monitors lassen sich diese Datensätze abschließend anzeigen.

17.3.6 Test des Föderierungssystems

Für den Test des Systems sind folgende Vorrausetzungen zu erfüllen:

- Der *mySQL*-Server und der O_2-Server müssen gestartet sein.
- Der Föderierungsgraph muß korrekt in der O_2-Datenbank angelegt sein.
- Die Datenbankapplikation, der Föderierungskern und der Datenbankagent müssen auf Grund der Verwendung von Unix-Pipes auf einem Rechner gestartet werden.

Das Föderierungssystem kann folgendermaßen gestartet werden, wobei die Reihenfolge der Programmaufrufe keine Rolle spielt:

- Start der Datenbankapplikation: `bookdb book_1`
- Start des Föderierungskerns: `main`
- Start des Datenbankagenten: `inagent book_2`

- Start des SQL-Monitors: `mysql book_2`.

Über die Applikation können nun Daten in die Datenbank `book_1` eingetragen werden. Über die Anweisung `select * from BOOK;` kann im SQL-Monitor kontrolliert werden, ob die Inhalte der Datensätze korrekt konvertiert und eingetragen wurden.

17.4 Ergebnisse

Die Überprüfung durch das Testsystem war erfolgreich, das bedeutet, es sind keine Fehler aufgetreten.

Der Agent der lokalen Datenbank `book_1` versendet nur Datensätze, die zuvor fehlerfrei in die Datenbank eingefügt wurden. Folglich empfängt der Föderierungskern nur gültige Datensätze. Die nach der Konvertierung an den Agenten der Datenbank `book_2` übermittelten Daten waren ebenfalls bezüglich des Aufbaus und Inhaltes gültig, was eine manuelle Überprüfung mittels des SQL-Monitors ergab. Die Graphalgorithmen, die auf dem in der O_2-Datenbank gespeicherten Graphen arbeiten, haben durchweg fehlerfreie Ergebnisse geliefert.

Daraus läßt sich folgern, daß der mit Hilfe des Schemaeditors spezifizierte und durch den Compiler übersetzte Föderierungsgraph korrekt ist.

Anhang A

Grammatik der Spezifikationssprache

A.1 Vokabular

In folgender Tabelle sind alle Schlüsselwörter der Spezifikationssprache aufgelistet.

Schlüsselwort	Beschreibung
SCHEMA_DEFINITION	Einleitung des Abschnittes zur Schemadefinition
END_SCHEMA_DEFINITION	Beendigung des Abschnittes zur Schemadefinition
SCHEMA	Einleitung der Definition eines Schemas
END_SCHEMA	Abschluß der Definition eines Schema
CLASS	Einleitung einer Klassendefinition
END_CLASS	Abschluß einer Klassendefinition
INVERSE	Inverses Attribut festlegen
POINTER	Zeigerdefinition
LIST	Listendefinition
SCHEMA_MAPPING	Einleitung des Abschnittes zur Abhängigkeitsdefinition
END_SCHEMA_MAPPING	Beendigung des Abschnittes zur Abhängigkeitsdefinition
BRANCH	Einleitung der Definition eines Pfades
END_BRANCH	Abschluß der Definition eines Pfades
MAP_SCHEMA	Definition von Abhängigkeiten auf Schemaebene
END_MAP_SCHEMA	Beendingung der Abhängigkeitsdefinition auf Schemaebene
MAP_TYPE	Definition von Abhängigkeiten auf Typebene
END_MAP_TYPE	Beendigung der Abhängigkeitsdefinition auf Typebene
MAP	Definition von Abhängigkeiten auf Attributebene
END_MAP	Beendingung der Abhängigkeitsdefinition auf Attributebene
PROC	Festlegung einer Konvertierungsfunktion
TO	Richtung $FS \rightarrow CS$ beim Mapping; Zieltyp bei Zeigerdefinition setzen
FROM	Richtung $CS \rightarrow FS$ beim Mapping
VIA	Bestimmung eines Schemas der Exportebene
OF	Elementtyp bei Listendefiniton
INTEGER	Standarddatentyp Integer
LONG	Standarddatentyp Long
BOOLEAN	Standarddatentyp Boolean
REAL	Standarddatentyp Real
DOUBLE	Standarddatentyp Double
CHARACTER	Standarddatentyp Character
STRING	Standarddatentyp String
DATE	Standarddatentyp Date
TIME	Standarddatentyp Time

A.2 Grammatik

BNF zur Spezifikationssprache

Standarddefinitionen

Terminals

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<name> ::= <letter>{<digit>|<letter>|_}
<remark> ::= #{<letter>|<digit>}
<schema_type_spec> ::= CS|IS|ES|IES|FS
<simple_type_spec> ::= STRING|CHAR|INTEGER|LONG|REAL|BOOLEAN|TIME|DATE
```

Ersetzungsregel für die Spezifikationssprache

```
<specification> ::= <schema_specification> <schema_mapping>
```

Regeln für den Schemadefinitionsblock

```
<schema_specification> ::= SCHEMA_DEFINITION <schema_def> {<schema_def>}
                           END_SCHEMA_DEFINITION

<schema_def> ::= SCHEMA <schema_specifier> <class_decl> {<class_decl>}
                 END_SCHEMA

<schema_specifier> ::= <schema_type>.<schema_name>
<schema_type> ::= <schema_es>|<schema_is>|<schema_ies>|<schema_fs>
<schema_name> ::= <name>
<schema_es>  ::= CS
<schema_is>  ::= IS
<schema_ies> ::= IES
<schema_fs>  ::= FS

<class_decl>   ::= <complex_decl> | <pointer_decl> | <compound_decl>

<complex_decl> ::= <complex_type> <class_name> <attribute> {<attribute>}
                   END_CLASS [<inverse_decl>]
<complex_type> ::= CLASS
<complex_name> ::= <name>
<class_name>   ::= <name>
<inverse_decl> ::= <inverse_type> <complex_name>.<attribute_name>
                                  <complex_name>.<attribute_name>
<inverse_type>   ::= INVERSE
<attribute_name> ::= <name>

<attribute>   ::= <simple_decl>|<generic_decl>

<simple_decl> ::= <simple_type> <simple_name>
<simple_type> ::= <simple_type_spec>
<simple_name> ::= <name>

<pointer_decl> ::= <pointer_type> <pointer_name> TO <attribute_name>
```

```
<pointer_type> ::= POINTER
<pointer_name> ::= <name>

<compound_decl> ::= <compound_type> <compound_name> OF <attribute_name>
<compound_type> ::= LIST
<compound_name> ::= <name>

<generic_decl> ::= <generic_type> <generic_name>
<generic_type> ::= <name>
<generic_name> ::= <name>
```

Regeln für den Mapping-Block

```
<schema_mapping> ::= SCHEMA_MAPPING <path_def> {<path_def>}
                     END_SCHEMA_MAPPING

<path_def> ::= BRANCH <path_name> <mapping_sources> END_BRANCH
<path_name> ::= <name>

<mapping_sources> ::= <mapping_schema_from>|<mapping_schema_to>

<mapping_schema_from> ::= MAP_SCHEMA <schema_fs> FROM <schema_cs> VIA
                                     <schema_es>|<schema_ies>
                                     {<mapping_type_from>}
                          END_MAP_SCHEMA

<mapping_schema_to> ::= MAP_SCHEMA <schema_fs> TO <schema_cs> VIA
                                   <schema_is>|<schema_ies>
                                   {<mapping_type_to>}
                        END_MAP_SCHEMA

<mapping_type_from> ::= MAP_TYPE <map_select>
                                 FROM <map_select>
                                 VIA <map_select>
                                 {<map_from>}
                        END_MAP_TYPE

<mapping_type_to> ::= MAP_TYPE <name>
                               TO <name>
                               VIA <name>
                               {<map_to>}
                      END_MAP_TYPE

<map_select> ::= <name>

<map_from> ::= <map_from_simple>|<map_from_nest>|<map_from_proc>
<map_to>   ::= <map_to_simple>|<map_to_nest>|<map_to_proc>

<map_from_simple> ::= MAP <map_select>
                          FROM <map_select>
                          VIA  <map_select>
                      END_MAP

<map_from_nest> ::= MAP <map_select>
```

```
                        FROM <map_select>,<map_select>{,<map_select>}
                        VIA  <map_select>
                        <proc_decl>
                    END_MAP

<map_from_proc> ::= MAP <map_select>
                        FROM <map_select>
                        VIA  <map_select>
                        <proc_decl>
                    END_MAP

<map_to_simple> ::= MAP <map_select>
                        TO  <map_select>
                        VIA <map_select>
                    END_MAP

<map_to_nest> ::= MAP <map_select>,<map_select>{,<map_select>}
                      TO  <map_select>
                      VIA <map_select>,<map_select>{,<map_select>}
                      <proc_decl>
                  END_MAP

<map_to_proc> ::= MAP <map_select>
                      TO  <map_select>
                      VIA <map_select>
                      <proc_decl>
                  END_MAP

<proc_decl> ::= PROC <proc_name>
<proc_name> ::= <name>
```

Anhang B

Erläuterungen zur lexikalischen Analyse

B.1 Der Quelltext

Für das Einlesen eines Quelltextes stellt Eli ein eigenständiges Modul bereit, welches den Quelltext verarbeitet und später auch für weitere Module zur Verfügung stellt. Dieses Modul braucht in der Regel nicht näher spezifiziert werden, denn es wird von Eli bei Bedarf selbstständig aus der Bibliothek abgeleitet.

Es sei allerdings die Funktionsweise kurz erklärt: Die Operation `initBuf` initialisiert dieses Modul und es bekommt zwei Parameter übergeben, den Dateinamen und einen Zeiger für diese Datei. Als Rückgabe von `initBuf` erhält das System `TEXTSTART`, die Adresse des ersten Buchstabens der Datei. Falls die Datei leer sein sollte, so addressiert `TEXTSTART` das Nullzeichen (ASCII NUL). Ansonsten garantiert `initBuf`, daß die Zeichenkette, die von `TEXTSTART` addressiert ist, mindestens eine vollständige Zeile inklusive dem abschließenden Newline-Character enthält.

Wenn der Prozeß alle Zeilen, die sich im Speicher befinden, abgearbeitet hat, so wird mittels der Methode `refillBuf` das nachfolgende Quellprogrammstück eingelesen.

B.2 Der Zwischencode

Der Syntax-Code, der für jedes Basissymbol berechnet wird, ist ausschließlich für die Zusammenarbeit der lexikalischen Analyse und der syntaktischen Analyse notwendig. Er wird von Eli intern zur Verfügung gestellt und ist deshalb nicht von Interesse und braucht auch nicht näher spezifiziert werden.

Jedes erkannte und voneinander unterscheidbare `name` muß von Eli mit einem eindeutigen Wert für das *wahre Attribut* versehen werden. Wenn derselbe `name` verwendet wird, so muß auch derselbe Wert zugewiesen werden. Da im späteren Verlauf diese Werte unter Umständen benutzt werden, so muß der Wert kompatibel zu den Modulen sein, die ihn benutzen. So z.B. gibt es ein Modul welches Zusammenhänge zwischen Definition und Gebrauch von `names` herstellen kann. Dieses Modul erwartet also z.B., daß jedes `name` durch eine eindeutige und kleine Integerzahl repräsentiert wird.

B.3 Das Scannen

Die in Abschnitt 6.2 erklärten *Abgrenzer*, die in dieser Sprache verwendet werden, sind bereits als Literale in der Grammatik spezifiziert worden. Eli extrahiert diese Literale aus der Grammatik und somit brauchen sie an dieser Stelle nicht noch einmal explizit definiert werden. In Kapitel 9 wurde bereits aufgeführt, daß in der Sprache drei verschiedene *Nicht-Literale* als Basissymbole verwendet werden, `name`, `schema_type_spec` und `simple_type_spec`. Diese müssen, genauso wie das Aussehen der Kommentare, genau spezifiziert werden.

Wie in Kapitel 9 ebenfalls schon erwähnt wurde, stellt Eli eine Bibliothek von vordefinierten Beschreibungen für Kommentare und *Basissymbol*-Formate zur Verfügung. Diese Bibliothek stellt nicht nur die unterschiedlichen Formate der Basissymbole zur Verfügung, sondern liefert auch die Methoden, die notwendig sind, um den Wert des benötigten *wahren Attributes* zu berechnen. Diese Beschreibungen sind also auch ein Beispiel für das Spezifizieren eines Teilproblems durch Analogie. Diese Methode wird aber bei dieser Sprache nur teilweise bei dem Kommentarformat verwendet.

```
name : $[a-zA-Z][a-zA-Z_0-9]* [mkidn]
```

Diese Spezifikation definiert das Basissymbol `name` , welches auf der linken Seite des Doppelpunktes steht. In Form eines regulären Ausdruckes wird definiert, welche Zeichen ein `name` beinhalten darf. In diesem Falle beginnt eine `name` immer mit einem Buchstaben, der entweder klein oder groß geschrieben werden darf. Darauf darf eine beliebige Anzahl Buchstaben oder auch von Dezimalziffern folgen und der Unterstrich "_" ist ebenfalls zugelassen. Dies entspricht fast der Definition eines C-Identifiers. Allerdings läßt ein C-Compiler auch den Unterstrich als erstes Zeichen zu. Aus diesem Grunde konnte hier nicht die vordefinierte Beschreibung `C_IDENTIFIER` verwendet werden. Sobald nun also eine solche Zeichenfolge erkannt wird, wird der Prozessor `mkidn` aufgerufen, der ein entsprechendes *wahres Attribut* zur Verfügung stellt.

Die in dieser Sprache zugelassenen Kommentare sind denen von awk-Skripten gleich. In diesem Fall kann also eine vordefinierte Spezifikation benutzt werden:

```
AWK_COMMENT
```

Kommentare unterscheiden sich bei der Spezifizierung dadaurch, daß sie kein Basissymbol getrennt durch einen Doppelpunkt zu ihrer linken Seite stehen haben. Solche Zeichenketten werden demnach ebenfalls erkannt und als unwichtig verworfen. Um die Verwendung von vordefinierten Spezifikationen zu verdeutlichen sei hier die äquivalente normale Schreibweise aufgeführt:

```
$# (auxEOL)
```

Der reguläre Ausdruck erkennt also an einer beliebigen Stelle ein "#". In Klammern kann eine Methode angegeben werden, die ausgeführt wird, sobald ein Zeichen erkannt wird, welches nicht durch den regulären Ausdruck abgedeckt ist - in diesem Fall also sofort nach dem Erkennen eines einzelnen "#". In diesem Falle wird `auxEOL` aufgerufen. Hierbei handelt es sich um einen sogenannten Fremdscanner, von denen auch einige durch Eli zur Verfügung gestellt werden. Der Scanner wird mit einem Zeiger auf das erste Zeichen nach dem "#" gestartet und liefert einen Zeiger auf das erste Zeichen der nächsten Zeile zurück. Mit Hilfe solcher Fremdscanner können also Zeichenfolgen übersprungen werden, also für den eigentlichen Scanvorgang unsichtbar gemacht werden. Genau dies ist die Eigenschaft von Kommentaren.

Bei den zwei folgenden Basissymbolen handelt es sich um eine Auswahl an feststehenden Zeichenketten. In der Regel hätten diese ganz einfach in der Grammatik als Abgrenzer auftauchen können und eine erneute Spezifizierung an dieser Stelle wäre unnötig gewesen. Allerdings werden diese Symbole auch bei der Ausgabe benötigt und dementsprechend müssen im später generierten Syntaxbaum auch Berechnungen auf diesen Knoten durchgeführt werden. Damit diese aber auch im Syntaxbaum als Knoten auftauchen, müssen diese Symbole Basissymbole sein. Der einfachste Weg dies zu bewerkstelligen ist also diese in der Grammatik als Terminale einzuführen und an dieser Stelle explizit zu definieren.

```
schema_type_spec : $CS|ES|IS|IES|FS [mkidn]
simple_type_spec:  $STRING|CHAR|INTEGER|LONG|REAL|BOOLEAN|TIME|DATE
                   [mkidn]
```

Es werden die Basissymbole `schema_type_spec` und `simple_type_spec` spezifiziert. Diese beiden Namen stehen wieder durch einen Doppelpunkt getrennt auf der linken Seite. Mit Hilfe der von Eli unterstützten Form von regulären Ausdrücken werden hier also feste reguläre Ausdrücke, wie z.B. `CS` und `ES`, durch Oder-Verknüpfungen miteinander verbunden. Der Scanner erkennt also für ein `schema_type_spec`-Symbol ausschließlich die fünf verschiedenen Buchstabenkombinationen. Analoges gilt für die Spezifikation von `simple_type_spec`.

Da in diesem Falle auch die genaue Schreibweise überprüft werden soll, müssen sich diese Spezifikationen also strikt von der Spezifikation von `name` trennen. Denn ein `schema_type_spec` würde natürlich auch durch den regulären Ausdruck von `name` richtig erkannt. Dabei würde aber auf der Strecke bleiben, daß die Großschreibung gewährleistet werden soll und auch natürlich die genaue Rechtschreibung. Mit anderen Worten würde ein Tippfehler im Quellprogramm der Form “Es” oder “Ess” als richtig erkannt, weil die Spezifikation von `name` diese Zeichenketten erkennen würde. Es hätte aber explizit nur “ES” erkannt werden dürfen, welches nur durch die Spezifikation von `schema_type_spec` gewährleistet wird. Da alle obenstehenden Spezifikationen später für eine Eingabe einfach der Reihe nach durchgetestet werden, und der Scannprozeß beendet wird sobald ein regulärer Ausdruck die Eingabe erkennt, muß also die Spezifikation von `name` als letzte in der Definition stehen. Die Spezifikation des Kommentares dagegen kann an einer beliebigen Stelle stehen.

B.4 Identifier Table

Der Prozessor `mkidn` assoziiert zu jedem voneinander unterschiedlichen Namen, die in B.3 spezifiziert sind, eine eindeutige Integerzahl. Dazu benutzt der Prozessor eine Hashtabelle, um zu überprüfen, ob der Name bereits zuvor schon einmal erkannt worden ist. Er speichert die Stringrepäsentierung des Namens einmal in einem großen Stringarray. Die Indexposition in diesem Array bildet dann den Wert des *wahren Attributes*.

B.5 Die Spezifikationsdatei für die lexikalische Analyse

Eine Datei vom Typ `.gla` beschreibt in Eli die lexikalische Struktur von Kommentaren und die Basissymbole, die nicht als Literale in der Grammatik erwähnt sind. Eli benutzt diese Informationen zusammen mit der Beschreibung der Literale, die aus der Grammatik abgeleitet werden können, um einen Scanner zu konstruieren.

```
/* Datei: sprache.gla */

schema_type_spec : $CS|ES|IS|IES|FS [mkidn]
simple_type_spec : $STRING|CHAR|INTEGER|LONG|REAL|BOOLEAN|TIME|DATE
                   [mkidn]
name             : $[a-zA-Z][a-zA-Z_0-9]* [mkidn]

                   $# (auxEOL)

/* Ende von sprache.gla */
```

Anhang C

Erläuterungen zur syntaktischen Analyse

C.1 `name` in verschiedenen Kontexten

Die Abbildung C.1 führt alle notwendigen Umbenennungen für die Grammatik auf, die notwendig sind, damit der Syntaxanalysierer von Eli selbstständig aus der Grammatik abgeleitet werden kann.

```
schema_name_def : name .
type_name_use : name .
type_name_def : name .
attribute_name_use : name.
attribute_name_von_use : name .
attribute_name_nach_use : name .
type_name_von_use : name .
type_name_nach_use : name .
attribute_name_def : name .
path_name_def : name .
proc_name_def : name .

type_name_use_cs : type_name_use .
type_name_use_fs : type_name_use .
type_name_use_es : type_name_use .
type_name_use_is : type_name_use .
map_select_fs : attribute_name_use .
map_select_es : attribute_name_use .
map_select_is : attribute_name_use .
map_select_cs : attribute_name_use .
map_select_multi_cs : attribute_name_use .
map_select : attribute_name_use .
map_select_multi_fs : attribute_name_use .
map_select_multi_is : attribute_name_use .
```

Abbildung C.1: Redefinition von `names` in verschiedenen Kontexten.

Entsprechende Spezifikationen müssen natürlich auch für die Symbole vorgenommen werden , die nicht `names` darstellen. Dies betrifft demnach noch die folgenden beiden Symbole:

```
schema_type_def : schema_type_spec .
simple_type : simple_type_spec .
```

C.2 Die Spezifikationsdatei für die syntaktische Analyse

Eine Datei vom Typ `.con` spezifiziert in Eli die Grammatik. Diese Information wird von Eli benutzt, um den Parser aufzubauen und die unterschiedlichen Basissymbole zu definieren. Diese Datei beinhaltet die Grammatik, die in Kapitel 6.3 beschrieben worden ist und die Definitionen aus Abbildung C.1 und ist in Abschnitt C.3 vollständig aufgeführt.

C.3 Die Grammatik in Eli-Notation

```
/*
#
# ELI Notation der Spezifikationssprache
#
/
specification : schema_specification schema_mapping .
/*
#
# Regeln fuer den Schemadefinitionsblock
#
/

schema_specification :  'SCHEMA_DEFINITION' schema_def+
                        'END_SCHEMA_DEFINITION' .
schema_def : 'SCHEMA' schema_specifier class_decl+ 'END_SCHEMA' .
schema_specifier : schema_type_def '.' schema_name_def .

class_decl : complex_decl / pointer_decl / compound_decl .

complex_decl : complex_type type_name_def attribute+ 'END_CLASS'
                   inverse_decl* .
complex_type : 'CLASS' .

inverse_decl : inverse_type type_name_von_use  '.' attribute_name_von_use
                        type_name_nach_use '.' attribute_name_nach_use .
inverse_type   : 'INVERSE' .

attribute   : simple_decl / generic_decl .

simple_decl : simple_type attribute_name_def .

pointer_decl : pointer_type type_name_def 'TO' type_name_use .
pointer_type : 'POINTER' .
```

```
compound_decl : compound_type type_name_def 'OF' type_name_use .
compound_type : 'LIST' .

generic_decl : type_name_use attribute_name_def .
```

/*
#
#
Regeln fuer den Mapping-Block
#
#
/

```
schema_mapping : 'SCHEMA_MAPPING' path_def+ 'END_SCHEMA_MAPPING' .
path_def : 'BRANCH' path_name_def mapping_schema_source 'END_BRANCH' .

mapping_schema_source : mapping_schema_from / mapping_schema_to .

mapping_schema_from : 'MAP_SCHEMA'
                        schema_specifier_fs 'FROM' schema_specifier_cs
                        'VIA' schema_specifier_es mapping_type_from+
                      'END_MAP_SCHEMA' .
mapping_schema_to   : 'MAP_SCHEMA'
                        schema_specifier_fs 'TO' schema_specifier_cs
                        'VIA' schema_specifier_is mapping_type_to+
                      'END_MAP_SCHEMA' .

mapping_type_from: 'MAP_TYPE' type_name_use_fs 'FROM' type_name_use_cs
                   'VIA' type_name_use_es map_from+ 'END_MAP_TYPE' .
mapping_type_to  : 'MAP_TYPE' type_name_use_fs 'TO' type_name_use_cs
                   'VIA' type_name_use_is map_to+ 'END_MAP_TYPE' .

map_from     : map_from_simple / map_from_nest / map_from_proc .
map_to       : map_to_simple / map_to_nest / map_to_proc .

map_from_simple : 'MAP' map_select_fs
                        'FROM' map_select_cs
                        'VIA'  map_select_es
                  'END_MAP' .
map_from_nest   : 'MAP' map_select_fs
                        'FROM' map_select_multi_cs ','
                               (map_select_multi_cs // ',')
                        'VIA'  map_select_es
                        proc_decl
                  'END_MAP' .

map_from_proc   : 'MAP' map_select_fs
                        'FROM' map_select_cs
                        'VIA'  map_select_es
                        proc_decl
                  'END_MAP' .

map_to_simple : 'MAP' map_select_fs
                      'TO'  map_select_cs
```

```
                      'VIA' map_select_is
                'END_MAP' .

map_to_nest   : 'MAP' fs_part
                      'TO' map_select_cs
                      'VIA' is_part
                      proc_decl
                'END_MAP' .
   fs_part: map_select_multi_fs ',' map_select_multi_fs
            (',' map_select_multi_fs)* .
   is_part: map_select_multi_is ',' map_select_multi_is
            (',' map_select_multi_is)* .

map_to_proc : 'MAP' map_select_fs
                    'TO'  map_select_cs
                    'VIA' map_select_is
                    proc_decl
              'END_MAP' .

proc_decl     : 'PROC' proc_name_def .
```

/*
##
Symbol-Renaming
##
/

```
simple_type             : simple_type_spec .
schema_type_def         : schema_type_spec .
schema_name_def         : name .
type_name_use           : name .
type_name_def           : name .
attribute_name_def      : name .
attribute_name_use      : name.
attribute_name_von_use  : name .
attribute_name_nach_use : name .
type_name_von_use       : name .
type_name_nach_use      : name .
path_name_def           : name .
proc_name_def           : name .
schema_specifier_fs     : schema_specifier .
schema_specifier_cs     : schema_specifier .
schema_specifier_es     : schema_specifier .
schema_specifier_is     : schema_specifier .
type_name_use_fs        : type_name_use .
type_name_use_cs        : type_name_use .
type_name_use_es        : type_name_use .
type_name_use_is        : type_name_use .
map_select_fs           : attribute_name_use .
map_select_es           : attribute_name_use .
map_select_is           : attribute_name_use .
map_select_cs           : attribute_name_use .
map_select_multi_cs     : attribute_name_use .
map_select_multi_fs     : attribute_name_use .
map_select_multi_is     : attribute_name_use .
```

Anhang D

Erläuterungen zu den Ausgabemustern

Wie bereits in Kapitel 13 beschrieben wurde, kann in Eli zur Ausgabe eines strukturierten Textes ein eigener Generator, der Pattern-Based Text Generator (kurz: PTG), benutzt werden. Die Struktur des Zieltextes wird hierbei durch eine Menge von Mustern beschrieben. PTG generiert daraus für jedes Muster eine eigene Funktion. Diese können dann aufgerufen werden, um eine Instanz der Zielstruktur zu konstruieren, die schließlich ausgegeben werden kann.

In diesem Anhang werden alle notwendigen Ausgabemuster, die in Abschnitt 13 eingeführt wurden, spezifiziert. Es wird genau auf die Zusammenhänge zwischen den Mustern und den Kontext zur Eingabesprache eingegangen.

D.1 Programmkopf

Der Programmkopf des Zielprogramms ist zunächst einmal ein feststehender Text, der für alle Quellprogramme gleich bleibt. An ihn schließen sich zwei große Textblöcke an, die Ausgaben für den Definitionsteil und die Ausgaben für den Mappingteil des Quellprogramms. Hierfür wird ein Muster namens `Table` spezifiziert (siehe Abbildung D.1). Zeichenketten, welche direkt ausgegeben werden sollen, werden in Anführungszeichen gesetzt. Ein Zeilenende muß hierbei explizit durch ein "\n" (newline) angegeben werden. Ansonsten werden alle Textsegmente hintereinander ausgegeben, auch wenn diese in verschiedenen Anführungszeichen stehen. Die beiden in der Spezifikation vorkommenden Dollarzeichen stehen als Platzhalter. Bei dem Aufruf, der aus diesem Muster generierten Methode `PTGTable`, werden demnach zwei Parameter erwartet. Diese Parameter müssen dann vom Typ PTGNode (siehe Abschnitt 14.1) sein. Auf diese Weise ist es möglich mehrfach geschachtelte Strukturen des Ausgabetextes zu konstruieren. Kommentare können auch in PTG-Spezifikationen, wie auch in allen anderen Eli-Spezifikationen, an jeder Stelle mit "`/* ... */`" eingefügt werden.

Das in Abbildung D.1 definierte Muster würde bis zu den Übergabeparametern also die in Abbildung D.2 gezeigte Ausgabe generieren.

D.2 Der Definitionsteil

In diesem Abschnitt werden alle Muster erläutert, die für die Ausgabe der Textsegmente notwendig sind, welche im Quellprogramm dem Definitionsblock entsprechen.

```
Table:
        "///////////////////\n"
        "//\n"
        "// Buildgraph.C\n"
        "//\n"
        "///////////////////\n\n"
        "#include \"build.H\"\n\n"
        "\n\n\n"
        "void BuildGraph ( PCGraph pg )\n"
        "{\n"
        "// require\n"
        "//assert (pg != NULL );\n"
        "\n\n\n"
        "//////////////////////\n"
        "//\n"
        "// (1)  Schemata anlegen\n"
        "//\n\n"
        $ /* Liste mit den Definitionen */
        "\n"
        "// Ende der Schemata\n"
        "\n\n\n"
        "///////////////////////\n"
        "//\n"
        "// Schemata verbinden\n"
        "//\n\n"
        $ /* Liste mit den Mappings */
        "// Ende der Schemata verbinden\n"
```

Abbildung D.1: Die Spezifikation für das Ausgabemuster `Table`.

```
///////////////////
//
// Buildgraph.C
//
///////////////////

#include "graph.H"


void BuildGraph ( PCGraph pg )
{
// require
//assert (pg != NULL );

/////////////////////
//
// (1)  Schemata anlegen
//
```

Abbildung D.2: Die Textausgabe, die von dem Muster `Table` generiert wird.

D.2.1 Schemadefinition

Das Muster für die Definition eines Schemas benötigt zunächst zwei Parameter, den Schemanamen und den Schematyp. Im späteren Anschluß an die Schemadefinition findet sofort die Definition der Klassen statt, die innerhalb des Schemas im Quellprogramm definiert sind. Aus diesem Grunde wird dem Muster als dritter Parameter auch schon die Liste von Klassendefinitionen übergeben, welche in späteren Symbolberechnungen zuvor ermittelt werden müssen. Eine weitere Funktionalität von PTG-Mustern sind die nummerierten Parameter, wie z.B. $1 oder $2. Die Nummerierung entspricht hierbei der Reihenfolge der Übergabe. Eine solche Nummerierung wird immer dann gebraucht, wenn Parameter z.B. mehrmals innerhalb des Musters benutzt werden. Man erspart sich dadurch die mehrmalige Übergabe desselben Parameters. Der Mustername ist in diesem Falle `SchemaList` und die Spezifikation ist in Abbildung D.3 zu sehen.

D.2.2 Klassendefinition

Die Spezifikation des Klassenmusters (siehe Abbildung D.4) erfolgt bis auf eine Erweiterung um einen Parameter analog. Für die Definition eines Klassennamens werden zunächst drei Parameter benötigt, der Schematyp, der Schemaname und der Klassenname. Im Falle der Klassendefinition erfolgt im unmittelbaren Anschluß die Definition der Attribute, die innerhalb der Klasse im Quellprogramm definiert wurden. Aus diesem Grunde wird hier, wie auch ähnlich bei der Schemadefinition, die Liste der Attributdefinitionen mit übergeben. Als weiterer Spezialfall bei der Klassendefinition sind noch die inversen Attribute zu erwähnen. Sie können im Quellprogramm im unmittelbaren Anschluß an eine Klassendefinition aufgeführt werden und erscheinen somit auch hier nach der Auflistung der Attribute, wenn also die Definition der Klasse abgeschlossen ist. Es wird ebenfalls eine Liste solcher Invers-Definitionen übergeben.

```
SchemaList:
        /* $1 = Schemaname
           $2 = Liste der Klassendefinitionen
           $3 = SchemaTyp */

        "// naechstes Schema\n\n"
        "PCSchema " $3 "_" $1 "=new CSchema(\"" $3 $1 "\",pg);\n"
        "\n\n"
        "// Schema mit Klassen fuellen\n"
        "\n\n"
        "//Typen\n"
        "BuildSimpleTypes ( " $3 "_" $1 " );\n\n"
        $2 /* Liste der Klassendefinitionen */
        "// Ende der Klassen\n"
```

Abbildung D.3: Die Spezifikation für das Muster `SchemList`.

```
ClassList:
        /* $1 = Klassenname
           $2 = Schemaname
           $3 = Liste der Attributedefinitionen
           $4 = SchemaTyp
           $5 = Liste von Inversdeklarationen */

        "// naechste Klasse\n\n"
        "PCComplexType " $4 "_" $2 "_" $1 "=new CComplexType(\"" $1
        "\", " $4 "_" $2 " );\n"
        "\n\n"
        "// Klasse mit Attributen fuellen\n"
        "\n"
        "//Attribute\n"
        "\n"
        $3 /* Liste der Attributdefinitionen */
        "\n"
        "// Ende der Attribute\n"
        $5 /* Liste der Invers-Definitionen */
```

Abbildung D.4: Spezifikation für das Muster `ClassList`.

```
InversList:
        /* $1 = SchemaType
           $2 = SchemaName
           $3 = Klassenname von
           $4 = Attributname von
           $5 = Klassenname nach
           $6 = Attributename nach */

        "// Inverses-Attribut\n"
        $1 "_" $2 "_" $3 "_" $4 "->SetInversAttr(" $1 "_" $2 "_" $5
        "_" $6 " );\n"
```

Abbildung D.5: Spezifikation des Musters `InversList`.

D.2.3 Inverse-Attribute-Definition

Die Inversen-Attribute, die im vorhergehenden Abschnitt erwähnt wurden, benötigen zur Spezifizierung sechs Übergabeparameter. Da die Definition nur innerhalb eines Schemas vorkommen kann, wird zunächst der Schematyp und der Schemaname übergeben. Des weiteren benötigt man dann den Klassennamen und den Attributnamen der jeweils invers zu verbindenden Attribute. In dieser Spezifikation sind keine weiteren Schachtelungsstrukturen mehr enthalten, es werden also keinerlei weitere Listen übergeben. Die Spezifikation in Abbildung D.5 ist deshalb leicht verständlich und auch die mehrfache Verwendung von Parametern wird hier anschaulich.

D.2.4 Attribut-Definition

Für die Definition eines Attributes werden zunächst als Paramter der Schematyp, der Schemaname, der Klassenname und natürlich der Attributname erwartet. Darüber hinaus muß auch der Attributtyp übergeben werden:

```
AttributeList:
        "PCAttribute " $3 "_" $4 "_" $5 "_" $2 " =
           new CAttribute ( \"" $2 "\", " $3 "_" $4 "
             ->GetType ( \"" $1 "\" ), " $3 "_" $4
               "_" $5 " );\n"
        "\n"
```

D.2.5 Pointer-, Listen- und Generic-Definition

Da die Spezifikationen von Pointern, Listen und generischen Attributen analog zu der Spezifikation von Attributen verläuft und keinerlei weitere Besonderheiten aufweisen, werden sie hier nur der Vollständigkeit in Abbildung D.6 zusammen aufgeführt.

D.2.6 Zusätzlich benötigte Muster

Für die spätere Generierung der Ausgabestruktur werden noch zwei weitere Muster benötigt. Sie alle spezifizieren Sequenzen von mehreren PTG-Ausgaben. In der Grammatik können innerhalb ei-

```
PointerList:
        "\n"
        "// Pointerdefinition\n"
        "\n"
        "PCPointerType " $1 "_" $2 "_" $3 " = new CPointerType ( "
        $1 "_" $2 " , " $1 "_" $2 "_" $4 " );\n"
        "\n"

CompoundList:
        "\n"
        "// Listendefinition\n"
        "\n"
        "PCCollectionType " $1 "_" $2 "_"$3 " =
            new CCollectionType (" $1 "_" $2 " , " $1 "_" $2 "_"
                                   $4 " );\n"
        "\n"

GenericList:
        "PCAttribute " $1 "_" $2 "_" $3 "_" $4 " =
            new CAttribute ( \"" $4 "\", " $1 "_" $2 "_" $5 " , "
                              $1 "_" $2 "_" $3 " );\n"
        "\n"
```

Abbildung D.6: Spezifikation der Muster für Pointer, Listen und Attribute.

nes Schemas nur drei verschiedene Typen von Definitionen vorgenommen werden, Klassen-, Pointer und Listdefinitionen. Um eine Ausgabe dieser drei Definitionen zu erreichen wird folgendes Muster spezifiziert:

```
CoPoLi:
        $ $ $
```

Ein ähnlicher Fall tritt bei der Definition von Attributen auf. Dort können zwei verschiedene Arten von Attributen definiert sein, *Simple-attributes* und *Generic-attributes*, die jeweils getrennt von einander erzeugten Ausgaben werden auch hier durch folgendes Muster aneinandergehängt:

```
SiGe:
        $ $
```

D.3 Der Mappingteil

D.3.1 Pre- und Post-Verbindungen

Im Abschnitt des Mappings können alle Schemata, Klassen, Attribute, jeweils schemaübergreifend mit einem gleichen Typ verbunden werden. Hierbei wird vor allem die Richtung der Verbindung unterschieden. Es werden deshalb jeweils für Schemaverbindungen, Klassenverbindungen, usw. zwei Muster spezifiziert, ein AddPre- und ein AddPost-Muster, welches jeweils die notwendige Richtung andeuten soll. Da sich diese Muster nur durch die Anzahl ihrer Parameter unterscheiden, die sie für die Definition benötigen, sei an dieser Stelle nur der Fall für die Attributverbindung in Abbildung D.7 aufgeführt. Für die Spezifikation von Schema- und Klassenverbindungen gilt analoges.

D.3.2 Parameterlisten- und Konverterdefinition

Im Mappingteil werden darüber hinaus noch Muster für die Definition von Konvertermethoden und für das Einrichten einer Parameterliste innerhalb von Attributobjekten benötigt (siehe Kapitel 6.1). Der Name der Konvertermethode wird dabei in ein Attributobjekt eingefügt. Demzufolge sind für die Spezifikation dieses Musters (siehe Abbildung D.8 fünf Parameter notwendig, vier davon definieren das Attribut, das letzte den Namen der Konvertermethode. Bei der Spezifikation für die dazugehörige Parameterliste (siehe Abbildung D.9) gilt ähnliches. Hier wird die Angabe von zwei Attributen benötigt, welches zu einer Angabe von acht Paramtern führt.

D.3.3 Weitere benötigte Muster

Die im weiteren noch verwendeten Muster zeichnen sich dadurch aus, daß sie alle nur Sequenzen von Ausgaben defnieren, also Spezifikationen sind, die in Abschnitt D.2.6 bereits erwähnt worden sind. Damit die Spezifikationen allerdings besser lesbar sind und auch die Semantik der Angaben ersichtlich wird, sind den Mustern unterschiedliche Namen gegeben worden, auch wenn sich die Syntax der Spezifikationen nicht unterscheidet. Da in der Quellsprache beim Mapping immer drei Schemata miteinander verbunden werden, so wird dies z.B. durch ein Muster mit dem Namen `MapSchemaFrom` wiedergegeben, welches auch vom Namen an das entsprechende Symbol in der Sprachgrammatik

```
AddPostAttribute:
        /* $1 : von-Schema-Type
           $2 : von-Schema-Name
           $3 : von-Type-Name
           $4 : von-Attribute-Name
           $5 : nach-Schema-Type
           $6 : nach-Schema-Name
           $7 : nach-Type-Name
           $8 : nach-Attribute-Name */

        $1 "_" $2 "_" $3 "_" $4 "->AddPostAttribute( "
          $5 "_" $6 "_" $7 "_" $8 " );\n"

AddPreAttribute:
        /* $1 : von-Schema-Type
           $2 : von-Schema-Name
           $3 : von-Type-Name
           $4 : von-Attribute-Name
           $5 : nach-Schema-Type
           $6 : nach-Schema-Name
           $7 : nach-Type-Name
           $8 : nach-Attribute-Name */

        $1 "_" $2 "_" $3 "_" $4 "->AddPreAttribute( "
          $5 "_" $6 "_" $7 "_" $8 " );\n"
```

Abbildung D.7: Muster für `AddPostAttribute` und `AddPreAttribute`.

```
SetConverter:
        /* $1 : SchemaType
           $2 : SchemaName
           $3 : Klassenname
           $4 : Attributname
           $5 : Convertername */

        $1 "_" $2 "_" $3 "_" $4 "->set_converter( new C" $5
        "Convert() );\n"
```

Abbildung D.8: Spezifikation für das Muster `SetConverter`.

```
AddPostParameter:
        /* $1 : SchemaType-mitte
           $2 : SchemaName-mitte
           $3 : Klassenname-mitte
           $4 : Attributname-mitte
           $5 : SchemaType-unten
           $6 : SchemaNamen-unte
           $7 : Klassenname-unten
           $8 : Attributname-unten */

        $1 "_" $2 "_" $3 "_" $4 "->AddPostParameter ("
        $5 "_" $6 "_" $7 "_" $8 ");\n"
```

Abbildung D.9: Spezifikation für das Muster `AddPostParameter`.

angelehnt ist (`mapping_schema_from`). Dieses Muster hat lediglich die Aufgabe drei Ausgaben miteinander zu verbinden, die Verbindung eines Komponentenschemas mit einem Exportschema und das Verbinden des Exportschemas mit dem föderierten Schema. Als dritte Ausgabe folgt die Liste mit den dazu passenden Typverbindungen. Also sieht die Spezifikation für dieses Muster folgendermaßen aus:

```
MapSchemaFrom:
        /* $1 Verbindet Cs mit Es
           $2 Verbindet Es mit Fs
           $3 Liste mit Type-Verbindungen */

        "// Verbindet die Schemata\n"
        $1
        $2
        "// Verbindet die Typen\n"
        $3
        "\n"
```

Im Falle dieses Musters ist die Benutzung von nummerierten Parametern nicht unbedingt notwendig. Für eine bessere Dokumentation der Spezifikationen ist sie allerdings durchaus sinnvoll. Somit kann in dem vorhergehenden Kommentar der Inhalt eines Parameters eindeutig beschrieben werden, was bei einer späteren Wiederverwendung der Spezifikationen zum besseren Verständis beiträgt.

Eine dem oben aufgeführten Muster analoge Spezifikation existiert demnach auch für die andere Mappingrichtung und somit für das entsprechende Symbol in der Grammatik mit dem Namen `mapping_schema_to`. Diese Spezifikation unterscheidet sich nur durch den Namen und den Kommentar, denn genau wie die obige Spezifikation verbindet auch `MapSchemaTo` drei Ausgaben miteinander:

```
MapSchemaTo:
        /* $1 Verbindet Is mit Cs
           $2 Verbindet Fs mit Is
           $3 Liste mit Type-Verbindungen */

        "// Verbindet die Schemata\n"
        $1
        $2
        "// Verbindet die Typen\n"
        $3
        "\n"
```

Verfolgt man nun die Grammtik so stellt man fest, daß es weitere ähnliche Symbole gibt, die bei der Generierung der Ausgabe eine solche Verkettung von mehreren Ausgaben benötigen. Da sich die dafür notwendigen Muster von obigen ebenfalls nur durch den Namen und eventuell durch die Anzahl der Parameter unterscheiden, seien hier nur ihre Namen aufgeführt, die sich leicht in den Kontext der Grammatik einordnen lassen, da ihre Namen ähnlich derer der zugehörigen Symbole der Grammatik sind:

```
MapTypeFrom, MapTypeTo,
MapFrom, MapTo
MapFromSimple, MapToSimple
MapFromProc, MapToProc
MapFromNest, MapToNest
```

An einigen Stellen bei Spezifikationen im Abschnitt 14 werden ebenfalls Sequenzen von Ausgaben benötigt, die sich allerdings von der Semantik der vorhergehenden unterscheiden und somit einen eigenen Namen bekommen:

```
Seq:
        $ $

Tripple:
        $ $ $
```

D.4 Spezifikationsdatei für die Ausgabemuster

Für die Ausgabe vorgesehene Spezifikationen von PTG-Mustern werden in Eli in Dateien vom Typ `.ptg` zusammengefaßt. Sämtliche oben erwähnten Musterspezifikationen werden für diesen Compiler in der Datei `ausgabe.ptg` beschrieben. Da die spätere Benutzung dieser Muster eng mit den Spezifikationen aus dem folgenden Kapitel zusammenhängen, wird an dieser Stelle davon Gebrauch gemacht die entsprechenden Dateien zu einer Datei zusammenzufassen. Demnach finden sich also die Spezifikationen für die Datei `ausgabe.ptg` in einer Datei mit dem Namen `ausgabe.fw`. Dateien vom Typ `.fw` kennzeichnen in Eli eine inhaltliche Zusammenlegung mehrerer Spezifikationsdateien.

Anhang E

Erläuterungen zu der Generierung der Ausgabe

E.1 Generierung ausgabefähiger Attribute

Aus Kapitel 14 ist bekannt, daß für die Generierung von Ausgaben mit Hilfe des PTG entsprechende Attribute vom Typ PTGNode vorliegen müssen. Diese PTG-Attribute müssen jedoch zuvor aus bereits bestehenden Attributen geschaffen werden, in dem bestehende Attribute umgewandelt werden. Da diese Umwandlungen für viele verschiedene Symbole der Grammatik durchgeführt werden müssen, wird ein generisches Symbol definiert (Entity), auf welchem die Berechnungen spezifiziert werden können. Später können dann die Symbole angegeben werden, auf denen die Berechnungen des generischen Symbols ausgeführt werden sollen. Die Attribute, auf denen diese Berechnungen ausgeführt werden sollen, sind alle `names`, wie durch die Grammatik beschrieben.

```
ATTR Sym: int;
SYMBOL Entity COMPUTE
  SYNT.Sym = CONSTITUENT name.Sym;
END;
```

Jedes dieser Entitys besteht also aus einem einzigen `name`, welches ein eindeutiges *Intrinsic-attribute* besitzt, daß von `mkidn` berechnet worden ist als `name` beim Einlesen erkannt worden ist. Eli erlaubt es jedem Terminal, also einem Blatt im Syntaxbaum, ein weiteres Integerattribut zu besitzen, welches dem Wert des vom Scanner berechneten Intrinsic-attributes entspricht. Der Wert dieses zusätzlichen Attributes wird also nie selbst gesetzt. Dieses Attribut ist durch den Namen *Sym* von Eli festgelegt. `SYNT.Sym = CONSTITUENT name.Sym` bedeutet nun also, daß das Attribut *Sym* eines *Entity* das selbe sein soll wie das Attribut *Sym* von `name`, welches sein Nachfolger im Syntaxbaum ist. `SYNT` spezifiziert hierbei ein synthetisiertes Attribut, welches demjenigen entspricht, mit dem die Berechnung gerade assoziiert ist - in diesem Falle Entity.

Die Benutzung von Daten von Nachfolgern in einem Syntaxbaum ist eine häufig vorkommende Operation im Compilerbau. Deshalb wird diese von Eli durch ein eigenes Spezifikationskonstrukt unterstützt: `CONSTITUENT`. Ein solcher CONSTITUENT-Ausdruck spezifiziert ein Nachfolgeattribut, welches als Quelle für die notwendigen Daten dient. Die genaue Vorgehensweise zur Verwendung dieses Spezifikationskonstruktes wird noch näher in Abschnitt E.2 erläutert.

```
SYMBOL schema_name_def INHERITS Entity END;
SYMBOL attribute_name_def INHERITS Entity END;
SYMBOL type_name_use INHERITS Entity END;
SYMBOL type_name_def INHERITS Entity END;
SYMBOL path_name_def INHERITS Entity END;
SYMBOL type_name_use_fs INHERITS Entity END;
SYMBOL type_name_use_es INHERITS Entity END;
SYMBOL type_name_use_is INHERITS Entity END;
SYMBOL type_name_use_cs INHERITS Entity END;
SYMBOL map_select_fs INHERITS Entity END;
SYMBOL map_select_es INHERITS Entity END;
SYMBOL map_select_is INHERITS Entity END;
SYMBOL map_select_cs INHERITS Entity END;
SYMBOL type_name_von_use INHERITS Entity END;
SYMBOL type_name_nach_use INHERITS Entity END;
SYMBOL attribute_name_von_use INHERITS Entity END;
SYMBOL attribute_name_nach_use INHERITS Entity END;
SYMBOL proc_name_def INHERITS Entity END;
SYMBOL map_select_multi_cs INHERITS Entity END;
SYMBOL map_select_multi_fs INHERITS Entity END;
SYMBOL map_select_multi_is INHERITS Entity END;
```

Abbildung E.1: Berechnung der PTG-Attribute für jeden notwendigen Knoten.

Die Bibliothek `LeafPtg` stellt Methoden zur Verfügung mit deren Hilfe man die Informationen in den Terminals (Blätter des Syntaxbaumes) in *PTGNodes* umwandeln kann. Zu diesem Zweck muß in der `.specs`-Datei der Compilerspezifikation das entsprechende Modul instanziiert werden:

```
$/Output/LeaftPtg.gnrc :inst
```

Es muß nun ein *PTGNode* generiert werden, der eine Zeichenkette eines Attributes mit dem Namen *Sym* repräsentiert, welches durch `mkidn` berechnet wurde. Eine solche Berechnung ist in der Bibliothek `LeafPtg` mit dem Symbol `IdPtg` assoziiert. Der so berechnete *PTGNode* ist der Wert des Ptg-Attributes desjenigen Symbols.

```
ATTR Ptg: PTGNode;
SYMBOL Entity INHERITS IdPtg END;
```

Diese beiden Spezifikationen für das Symbol Entity gewährleisten den gewünschten Effekt eines ausgabefähigen Attributinhaltes und somit werden alle Symbole der Grammatik, die für die Ausgabe benötigt werden und als Inhalt einen `name` haben, dieser Berechnung unterzogen. Die Abbildung E.1 zeigt die Gesamtheit dieser Spezifikation.

Somit erhalten alle die aufgeführten Knoten des Syntaxbaumes ein Attribut namens `Ptg`, dessen Wert den String des Knotens im Eingabetext repräsentiert.

Außer `name` sind in dieser Sprache noch zwei weitere Terminal-Attribute definiert: `simple_type` und `schema_type`. Diese beinhalten jedoch nicht einen `name`, sondern jeweils ein `sim-`

```
SYMBOL SimpleTypeComputation COMPUTE
  SYNT.Sym=CONSTITUENT simple_type_spec.Sym;
END;
SYMBOL SimpleTypeComputation INHERITS IdPtg END;
SYMBOL simple_type INHERITS SimpleTypeComputation END;

SYMBOL SchemaTypeComputation COMPUTE
  SYNT.Sym=CONSTITUENT schema_type_spec.Sym;
END;
SYMBOL SchemaTypeComputation INHERITS IdPtg END;
SYMBOL schema_type_def INHERITS SchemaTypeComputation END;
```

Abbildung E.2: Berechnungen der PTG-Attribute für die Knoten `simple_type` und `simple_type_def`.

```
ATTR SchemaNamePtg, SchemaTypePtg: PTGNode;

SYMBOL schema_def COMPUTE
  SYNT.SchemaNamePtg = CONSTITUENT schema_name_def.Ptg ;
  SYNT.SchemaTypePtg = CONSTITUENT schema_type_def.Ptg ;
END;
```

Abbildung E.3: Kopieren des Schemanamens und des Schematyps aus den Blättern unterhalb des Knotens `schema_def`.

`ple_type_spec` und `schema_type_spec`, welche das *wahre Attribut* und somit auch das Sym-Attribut beinhalten. Aus diesem Grunde können die obigen Berechnungen nun nicht auf diese Symbole angewendet werden. Für die Ausgabe sind sie allerdings ebenfalls erforderlich und somit müssen speziell für diese beiden Symbole eigene Berechnungen spezifiziert werden, die analog zu den vorhergehenden erfolgen (siehe Abbildung E.2).

E.2 Berechnungen für den Definitionsblock

In Abschnitt 14.3 wurde bereits diskutiert, daß für die Ausgabe von Klassendefinitionen die Informationen über Schemanamen und Schematyp im Syntaxbaum kopiert werden müssen. Die dabei zu Grunde liegende Vorgehensweise wurde dabei auch besprochen.

Von dort abgeleitet bedeutet dies, daß zunächst in dem Knoten `schema_def` zwei neue Attribute vom Typ `PTGNode` definiert werden, in die jeweils einfach das entsprechende PTG-Attribut von `schema_type_def` und `schema_name_def` kopiert wird. Die Spezifikation dazu zeigt Abbildung E.3.

Für die Ausgabe einer Klassendefinition würde dies im Prinzip reichen, da von einem Knoten des Syntaxbaumes auf Attribute des direkt über ihm liegenden Knotens zugegriffen werden kann. Da allerdings auch für die noch kommenden Ausgaben der Attributdefinitionen der Schematyp und der Schemaname notwendig sind (siehe Muster `PTGAttributeList` in Abschnitt D.2.4), müssen diese Daten auch in den Knoten `complex_decl` kopiert werden. Dazu werden Berechnungen auf jeden

Knoten `complex_decl` ausgeführt, welche die beiden Attribute `SchemaNamePtg` und `SchemaTypePtg` aus dem über ihnen liegenden Knoten `schema_def` kopieren:

```
SYMBOL complex_decl COMPUTE
  SYNT.SchemaNamePtg = INCLUDING schema_def.SchemaNamePtg;
  SYNT.SchemaTypePtg = INCLUDING schema_def.SchemaTypePtg;
END;
```

Das Spezifikationskonstrukt `INCLUDING` ermöglicht also den Zugriff auf oberhalb liegende Knoten, während `CONSTITUENT` auf unterhalb liegende Knoten Zugriff ermöglicht. In Verbindung mit der vorhergehenden Spezifikation aus Abbildung E.3 verdeutlicht die Grafik in Abbildung 14.1 auf Seite 60 diesen Zusammenhang. Bei den vorhin angesprochenen Attributdefinitionen wird neben dem Schemanamen und dem Schematyp auch noch der Klassenname benötigt. Aus der Grammatik läßt sich allerdings erkennen, daß sich das Symbol `simple_decl` aus dem Symbol `complex_decl` verzweigt. Der zugehörige Klassenname verzweigt sich auch hier im Syntaxbaum von demselben Knoten `complex_decl`. Somit tritt auch hier der Fall ein, daß bei den Berechnungen auf dem Knoten `simple_decl` nicht auf den Klassennamen, der sich in `type_name_use` befindet und in einem parallelen Zweig liegt, zugegriffen werden kann. Die Grafik in Abbildung 14.3 auf Seite 61 zeigt nochmals diesen Sachverhalt. Aus diesem Grunde muß auch hier zunächst der Klassenname in den gemeinsamen Knoten `complex_decl` kopiert werden:

```
ATTR ComplexNamePtg : PTGNode;
SYMBOL complex_decl COMPUTE
  SYNT.ComplexNamePtg = CONSTITUENT type_name_def.Ptg;
END;
```

Um später einfacher auf die Daten zugreifen zu können werden nun der Schemaname, der Schematyp und der Klassenname direkt in den Knoten `simple_decl` kopiert:

```
SYMBOL simple_decl COMPUTE
  SYNT.SchemaNamePtg = INCLUDING complex_decl.SchemaNamePtg;
  SYNT.SchemaTypePtg = INCLUDING complex_decl.SchemaTypePtg;
  SYNT.ComplexNamePtg = INCLUDING complex_decl.ComplexNamePtg;
END;
```

Neben dem Symbol `simple_decl` existiert auch noch ein weiteres Symbol, welches Attribute im Quelltext definieren kann: `generic_decl`. Da für dieses Symbol dieselben Informationen benötigt werden, um eine entsprechende Ausgabe zu generieren, werden auch in diesen Knoten die notwendigen Daten kopiert. Die dafür notwendige Spezifikation ist analog zu der für das Symbol `simple_decl` und wird deshalb hier nicht mehr aufgeführt.

Neben Klassen können in Schemata auch noch Pointer, Listen und inverse Attribute definiert werden. Für eine Ausgabe einer Pointerdefinition werden neben den direkt angegebenen Klassen- und Attributnamen auch wieder der Schematyp und der Schemaname benötigt, die im Syntaxbaum oberhalb von `pointer_decl` liegen. Da dieser Zweig parallel zu dem Zweig der Klassendefinition (`complex_decl`) liegt, liegen die notwendigen Daten ja bereits im Knoten `schema_def` bereit und werden für die einfachere Benutzung direkt in den Knoten `pointer_decl` kopiert:

```
SYMBOL SchemaTypeandName COMPUTE
  SYNT.SchemaTypePtg = CONSTITUENT schema_type_def.Ptg;
  SYNT.SchemaNamePtg = CONSTITUENT schema_name_def.Ptg;
END;

SYMBOL schema_specifier_fs INHERITS SchemaTypeandName END;
SYMBOL schema_specifier_cs INHERITS SchemaTypeandName END;
SYMBOL schema_specifier_is INHERITS SchemaTypeandName END;
SYMBOL schema_specifier_es INHERITS SchemaTypeandName END;
```

Abbildung E.4: Kopieren des Schemanamen und des Schematypen in einen im Graphen oberhalb liegenden Knoten.

```
SYMBOL pointer_decl COMPUTE
  SYNT.SchemaNamePtg = INCLUDING schema_def.SchemaNamePtg;
  SYNT.SchemaTypePtg = INCLUDING schema_def.SchemaTypePtg;
END;
```

Eine analoge Symbolberechnung findet demnach auch für die Listendefinition und für die inversen Attribute statt, also auf dem Knoten `compound_decl` und `inverse_decl` im Syntaxbaum. Da auch hier die Spezifikation identisch mit der vorhergehenden ist, seien sie nicht mehr aufgeführt.

Bei den zuletzt erwähnten Knoten werden lediglich die notwendigen Daten aus oberhalb liegenden Knoten *herunterkopiert*. Bei den zur Ausgabe führenden Berechnungen muß natürlich noch auf Knoten, die unterhalb im Syntaxbaum liegen, zugegriffen werden. Dies kann jedoch einfach mittels `CONSTITUENT` an den entsprechenden Stellen geschehen. Somit wird bei den Symbolberechnungen immer nur nach unten in den Syntaxbaum *gegriffen*, sodaß die Berechnungen im späteren Verlauf leichter zu lesen und zu verstehen sind.

Somit sind für den Definitionsblock die Berechnungen in soweit abgeschlossen, als daß in den zur Ausgabe führenden Knoten `complex_decl`, `pointer_decl`, `compound_decl`, `inverse_decl`, `simple_decl` und `generic_decl` die jeweils notwendigen Daten vorhanden sind, bzw. leicht auf die noch fehlenden zugegriffen werden kann.

E.3 Berechnungen für den Mappingblock

Aus Abschnitt 14.4 auf Seite 62 ist bekannt, daß für alle Ausgaben im Mappingteil der Schemaname und der Schematyp benötigt werden und daß diese wegen ihrer Position im Syntaxbaum zunächst in den oberhalb liegenden Knoten kopiert werden müssen. Die Abbildung E.4 zeigt die dazu notwendigen Spezifikationen auf.

Da diese Berechnungen für mehrere unterschiedliche Symbole durchgeführt werden müssen, macht man sich auch hier zunutze ein imaginäres Symbol zu spezifizieren und auf diesem die Berechnungen zu definieren. Anschließend werden dann nur noch die unterschiedlichen Symbole angegeben, für welche die Berechnungen ausgeführt werden sollen.

Im weiteren wird nun in die beiden Mappingrichtungen unterschieden.

```
ATTR FSSchemaTypePtg, FSSchemaNamePtg,
     ESSchemaTypePtg, ESSchemaNamePtg,
     CSSchemaTypePtg, CSSchemaNamePtg,
     FSTypeNamePtg, ESTypeNamePtg, CSTypeNamePtg : PTGNode;

SYMBOL mapping_schema_from COMPUTE
SYNT.FSSchemaTypePtg=CONSTITUENT schema_specifier_fs.SchemaTypePtg;
SYNT.FSSchemaNamePtg=CONSTITUENT schema_specifier_fs.SchemaNamePtg;
SYNT.ESSchemaTypePtg=CONSTITUENT schema_specifier_es.SchemaTypePtg;
SYNT.ESSchemaNamePtg=CONSTITUENT schema_specifier_es.SchemaNamePtg;
SYNT.CSSchemaTypePtg=CONSTITUENT schema_specifier_cs.SchemaTypePtg;
SYNT.CSSchemaNamePtg=CONSTITUENT schema_specifier_cs.SchemaNamePtg;
END;
```

Abbildung E.5: Kopieren von verschieden Schemanamen und Schematypen in den im Graphen oberhalb liegenden Knoten.

E.3.1 Mapping-From

Abschnitt 14.4.1 auf Seite 63 beschreibt, daß beim Mapping in dieser Sprache grundsätzlich drei Objekte vom gleichen Typ miteinander verbunden werden. Den Anfang bildet dabei das Verbinden von Schemata, innerhalb dieser folgen Pointer, Listen und Klassen und innerhalb von Klassen wiederum noch Attribute. In allen Fällen werden die Namen und Typen der drei beteiligten Schemata benötigt. Im Fall der Attributverbindungen darüber hinaus auch die Namen der zugehörigen Klassen. Wenn durch den Scanner das Symbol `mapping_schema_from` erkannt wird, so ist an den Verbindungen neben dem föderierten und dem Komponentenschema noch ein Exportschema beteiligt. Diese Namen und Typen müssen zunächst, wie in Abbildung E.5 gezeigt, für die weitere Verwendung in den gemeinsamen Knoten `mapping_schema_from` kopiert werden.

Hierzu werden für jeden Schematyp jeweils ein Attribut für den Typ und den Namen des Schemas angelegt. Diese Attribute sind nach wie vor alle vom Typ `PTGNode` und deren Inhalte werden aus den dazu passenden, bereits berechneten PTG-Attributen kopiert. Es wurde ebenfalls schon beschrieben, daß im Syntaxbaum (Abbildung 14.5 auf Seite 64) dem Knoten `mapping_schema_from` der Knoten `mapping_type_from` folgt. In diesen werden zunächst aus Gründen der besseren Übersicht die Schemanamen und Schematypen von vorhin kopiert. Zusätzlich dazu verfügt der Knoten `mapping_type_from` über drei Zweige, in denen jeweils die in die Schemata passenden Klassen angegeben sind. Auch diese Informationen müssen in dem Knoten `mapping_type_from` gesammelt werden, um diese später verwenden zu können. Es werden also drei weitere Attribute angelegt und aus den drei Zweigen jeweils das richtige PTG-Attribut kopiert. Die dazu notwendigen Spezifikationen lassen sich der Abbildung E.6 entnehmen.

Die `INCLUDING`-Konstrukte kopieren wie gewohnt die Daten aus dem darüber liegenden Knoten und die drei `CONSTITUENT`-Konstrukte sammeln aus den drei Zweigen die nötigen Daten zusammen.

Die bislang aufgeführten Knoten des Syntaxbaumes führen allerdings zu keinerlei Ausgaben. Sie beinhalten lediglich dazu notwendige Informationen. Die Knoten `map_from_simple`, `map_from_proc` und `map_from_nest` dagegen führen später zu Ausgaben. Damit bei den dazu später notwendigen Funktionsaufrufen die Parameterübergabe so einfach wie möglich ist, werden in diese Knoten nun die gesammelten Daten über Schemanamen, Schematypen und Klassennamen

```
SYMBOL mapping_type_from COMPUTE
SYNT.FSSchemaTypePtg=INCLUDING mapping_schema_from.FSSchemaTypePtg;
SYNT.FSSchemaNamePtg=INCLUDING mapping_schema_from.FSSchemaNamePtg;
SYNT.ESSchemaTypePtg=INCLUDING mapping_schema_from.ESSchemaTypePtg;
SYNT.ESSchemaNamePtg=INCLUDING mapping_schema_from.ESSchemaNamePtg;
SYNT.CSSchemaTypePtg=INCLUDING mapping_schema_from.CSSchemaTypePtg;
SYNT.CSSchemaNamePtg=INCLUDING mapping_schema_from.CSSchemaNamePtg;
SYNT.FSTypeNamePtg  =CONSTITUENT type_name_use_fs.Ptg;
SYNT.ESTypeNamePtg  =CONSTITUENT type_name_use_es.Ptg;
SYNT.CSTypeNamePtg  =CONSTITUENT type_name_use_cs.Ptg;
END;
```

Abbildung E.6: Kopieren von Schemanamen und Schematyp aus einem im Graphen oberhalb liegenden Knoten in Verbindung mit dem Kopieren von Typnamen aus im Graphen unterhalb liegenden Knoten.

```
SYMBOL map_from_simple COMPUTE
SYNT.FSSchemaTypePtg = INCLUDING mapping_type_from.FSSchemaTypePtg;
SYNT.FSSchemaNamePtg = INCLUDING mapping_type_from.FSSchemaNamePtg;
SYNT.ESSchemaNamePtg = INCLUDING mapping_type_from.ESSchemaNamePtg;
SYNT.ESSchemaTypePtg = INCLUDING mapping_type_from.ESSchemaTypePtg;
SYNT.CSSchemaTypePtg = INCLUDING mapping_type_from.CSSchemaTypePtg;
SYNT.CSSchemaNamePtg = INCLUDING mapping_type_from.CSSchemaNamePtg;
SYNT.FSTypeNamePtg = INCLUDING mapping_type_from.FSTypeNamePtg;
SYNT.ESTypeNamePtg = INCLUDING mapping_type_from.ESTypeNamePtg;
SYNT.CSTypeNamePtg = INCLUDING mapping_type_from.CSTypeNamePtg;
END;
```

Abbildung E.7: Kopieren von Attributen aus einem im Graphen oberhalb liegenden Knoten.

in diese Knoten propagiert. Die Spezifikationen dazu sind folglich leicht aus dem vorhergehenden Absatz abzuleiten und werden in Abbildung E.7 gezeigt.

Die Spezifikation für den Knoten `map_from_proc` ist analog und wird nicht mehr aufgelistet.

E.3.2 Map-From-Nest

Der Abschnitt 14.4.2 beschreibt, daß für die Berechnungen im Knoten `map_from_nest` eine zusätzliche Spezifikation notwendig ist. Dabei handelt es sich um das zusätzliche Kopieren des Attributnamens aus dem beteiligten Exportschemas. Dieses muß ebenfalls in den übergeordneten Knoten `map_from_nest` kopiert werden. Zusammen mit den Daten über Schemanamen, Schematypen und Klassennamen ergibt dies die in Abbildung E.8 gezeigte, um ein Attribut erweiterte Spezifikation.

Diese so gewonnenen Daten müssen nun wie schon mehrmals durchgeführt in jeden Knoten vom Typ `map_select_multi_cs` propagiert werden, damit die späteren Funktionsaufrufe eine möglichst übersichtliche und einfache Struktur besitzen. Eine Spezifikation dazu erfolgt analog zu den bisher genannten Kopierspezifikationen und wird deshalb nicht mehr explizit aufgeführt.

```
ATTR ESAttributeNamePtg : PTGNode;

SYMBOL map_from_nest COMPUTE
SYNT.FSSchemaTypePtg = INCLUDING mapping_type_from.FSSchemaTypePtg;
SYNT.FSSchemaNamePtg = INCLUDING mapping_type_from.FSSchemaNamePtg;
SYNT.ESSchemaNamePtg = INCLUDING mapping_type_from.ESSchemaNamePtg;
SYNT.ESSchemaTypePtg = INCLUDING mapping_type_from.ESSchemaTypePtg;
SYNT.CSSchemaTypePtg = INCLUDING mapping_type_from.CSSchemaTypePtg;
SYNT.CSSchemaNamePtg = INCLUDING mapping_type_from.CSSchemaNamePtg;
SYNT.FSTypeNamePtg = INCLUDING mapping_type_from.FSTypeNamePtg;
SYNT.ESTypeNamePtg = INCLUDING mapping_type_from.ESTypeNamePtg;
SYNT.CSTypeNamePtg = INCLUDING mapping_type_from.CSTypeNamePtg;
SYNT.ESAttributeNamePtg = CONSTITUENT map_select_es.Ptg;
END;
```

Abbildung E.8: Kopieren von Schemanamen und Schematypen in den Knoten `map_from_nest` aus dem im Graphen oberhalb liegenden Knoten `mapping_type_from`. Gleichzeitig wird ein Attributname aus einem im Graphen unterhalb liegenden Knoten ebenfalls kopiert.

E.3.3 Map-To-Nest

In Abschnitt 14.4.4 ab Seite 66 wird bereits ausführlich die Problematik beim Map_to_nest beschrieben. Dabei geht es hauptsächlich um die Verwendung des Moduls `LidoList` aus der Eli-Bibliothek. Es wird beschrieben, daß für alle Knoten `map_select_multi_fs` der Inhalt ihres Ptg-Attributes in ein neues Attribut mit dem Namen PTGNodeElem kopiert werden muß. Dieses Attribut ist natürlich auch vom Typ `PTGNode`:

```
ATTR PTGNodeElem : PTGNode;

SYMBOL map_select_multi_fs COMPUTE
  SYNT.PTGNodeElem = THIS.Ptg;
END;
```

Nachfolgend wurde im Abschnitt 14.4.4 auch beschrieben, daß die LidoList im Knoten `fs_part` angelegt werden muß. Dazu muß definiert werden welches Symbol die Wurzel der Liste ist und welche Knoten die Listenelemente beinhalten:

```
SYMBOL fs_part             INHERITS PTGNodeListRoot   END;
SYMBOL map_select_multi_fs INHERITS PTGNodeListElem   END;
```

Da nun die so gewonnene Liste im Knoten `fs_part` nicht abgebaut werden kann, muß diese in einen anderen Knoten kopiert werden. Für diesen Fall kann keine Symbolberechnung, wie wir sie bislang verwendet haben, benutzt werden, sondern es muß eine sogenannte `RULE`-Berechnung durchgeführt werden.

```
RULE: map_to_nest::='MAP' fs_part 'TO' map_select_cs 'VIA' is_part
proc_decl 'END_MAP'
  COMPUTE is_part.PTGNodeList = fs_part.PTGNodeList;
END;
```

Es läßt sich dabei erkennen, daß für die Spezifikation dieser Berechnung im Prinzip die komplette Spezifikation aus der Grammatik übernommen wird, für die eine Berechnung stattfinden soll, wenn diese erkannt wird. Der Vorteil dieser Methode ist, daß man auf alle vorkommenden Symbole direkt zugreifen kann. Diese Möglichkeit war bei der Verwendung der `SYMBOL`-Berechnungen ja nicht uneingeschränkt möglich. Ein sehr großer Nachteil bei der Verwendung dieser Berechnung ist die mangelnde Wartbarkeit. Wenn zu einem späteren Zeitpunkt die Grammatik der Sprache für das Symbol `map_to_nest` abgeändert wird, so muß auch an dieser Stelle bei den Berechnungen die `RULE` nachgebessert werden. Solange die Symbolnamen nicht verändert werden, ist eine solche Abänderung für die `SYMBOL`-Berechnungen eher ungefährlich. Es ist also bei der Verwendung Vorsicht geboten.

Das Kopieren der Liste erfolgt dann durch `COMPUTE is_part.PTGNodeList = fs_part.PTGNodeList;`. Die Liste wird im Knoten also unter dem automatisch ausgewählten Attributnamen `PTGNodeList` abgelegt. Kopiert wird die Liste hierbei durch Referenzierung, das bedeutet, nicht die Inhalte werden kopiert, sondern nur der interne Zeiger auf diese Liste.

Zum Schluß wird die Liste des Knotens `is_part` für den schrittweisen Abbau vorbereitet:

```
SYMBOL is_part              INHERITS PTGNodeDeListRoot END;
SYMBOL map_select_multi_is  INHERITS PTGNodeDeListElem END;
```

Jetzt wird auch klar warum in einem Knoten nicht gleichzeitig eine LidoList konstruiert und gleichzeitig für den Abbau spezifiziert werden kann. Im übrigen macht dies auch nicht unbedingt Sinn, denn wenn man auf den Knoten mit der Liste zugreifen kann, um die Liste abzubauen, dann könnte man auch auf alle darunterliegenden Knoten im Syntaxbaum zugreifen und eine extra Liste wäre unnötig, was sich ja auch bei der jetzigen Vorgehensweise bestätigt hat. Die Knoten `map_select_multi_is` beinhalten also nun ein Attribut mit dem Namen `PTGNodeElem`, welches das jeweils passende Ptg-Attribut aus der Liste enthält und auf welches nun bei den Funktionsaufrufen für die Ausgabengenerierung direkt zugegriffen werden kann.

Die restlichen Spezifikationen, die noch für das Map-To-Nest notwendig sind, ähneln denen des Map-From-Nest. Es werden später für jeden Knoten `map_select_multi_is` Ausgaben generiert. Für diese sind die Schematypen, die Schemanamen und die Klassennamen notwendig, ebenso wie der Attributname im Komponentenschema. Dazu werden im Knoten `map_to_nest` die Informationen über die Schematypen, Schemanamen und Klassennamen aus dem über ihm liegenden Knoten `mapping_type_to` kopiert und die Information über das eine Komponentenschemaattribut aus dem Knoten `map_select_cs` geholt. Damit diese Daten später in jedem `map_select_multi_is` leicht zur Verfügung stehen und die notwendigen Funktionsaufrufe leichter zu verfolgen sind, werden diese gesammelten Daten in jeden Knoten `map_select_multi_is` hineinkopiert. Die dazu notwendigen Spezifikationen seien hier nicht mehr aufgeführt, da sie denen für das Map-From-Nest sehr ähnlich sind und sich nur durch die benutzten Symbolnamen unterscheiden.

Zum Abschluß fehlt noch die Spezifikation, die das Module `LidoList` instanziiert. Da LidoListen grundsätzlich von einem beliebigen Typ sein können, muß bei der Instanziierung der genaue Typ angegeben werden. Da dies auch nicht-eli-eigene Typen sein können, ist auch die Angabe einer C-Deklarationsdatei notwendig, in dem der Typ spezifiziert ist. Die hier verwendeten Listenelemente

sind vom Typ PTGNode. Allerdings ist nirgendwo dokumentiert wie die zugehörige Deklarationsdatei in Eli benannt ist, noch in welchem Eli-Verzeichnis sich diese befindet, denn diese Informationen werden in der Regel nicht benötigt, da diese nur eli-intern Verwendung finden. Da das Modul, welches die PTGNodes zur Verfügung stellt, ebenfalls instanziiert worden ist und die damit verbundenen eli-internen C-Codes und Deklarationsdateien generisch erzeugt worden sind, war eine konkrete Suche zunächst erfolglos. Letztendlich stellt sich heraus, daß diese Datei den Namen `ptg_gen` (ohne Dateiendung) trägt. Somit kann die gewünschte Listenform wie folgt nutzbar gemacht werden:

```
$/Adt/LidoList.gnrc +instance=PTGNode +referto=ptg_gen :inst
```

Das Modul LidoList definiert bei der Instanziierung automatisch eine Variable, die das leere Listenelement kennzeichnet. Der Variablenname ist auch nicht explizit dokumentiert und lautet in diesem Falle `NoPTGNode`. Diese Variable muß für eine spätere Verwendung definiert werden, da das Modul LidoList nicht wissen kann wie ein leerer PTGNode definiert ist. Eine solche Definition muß in Eli in C-Form und in einer Datei vom Typ `.head` vorgenommen werden und sieht deshalb wie folgt aus:

```
#define NoPTGNode PTGNULL
```

E.4 Spezifikationsdateien für die Berechnungen

Eine Datei vom Typ `.lido` beschreibt die notwendigen Symbol- und Rule- Berechnungen. Für diesen Compiler sind sämtliche zuvor aufgeführten Spezifikationen in der Datei `name.lido` zusammengefaßt.

Die Instanziierung der LidoList wird, genau wie die Instanziierung von `LeafPtg` in der Datei `sprache.specs` angegeben.

Da die Definition der Variablen `NoPTGNode` in eine Datei vom Typ `.head` vorgenommen werden muß, wird die Datei `name.head` angelegt. Diese Datei muß dann ebenfalls in der globalen Datei `sprache.specs` eingetragen werden, damit sie bei der Compilergenerierung mit eingebunden wird.

E.5 Die Ausgabe

E.5.1 Programmkopf

```
SYMBOL specification COMPUTE
  PTGOut(
    PTGTable(
        CONSTITUENTS schema_def.Ptg WITH (PTGNode, PTGSeq,
                                          IDENTICAL, PTGNull),
        CONSTITUENTS path_def.Ptg   WITH (PTGNode, PTGSeq,
                                          IDENTICAL, PTGNull)));
END;
```

Die Wurzel des Baumes, also das Symbol `specification` führt zu der Ausgabe der PTG-Struktur. Für diese Ausgabe wird die Funktion PTGTable aufgerufen. Diese ist aus Kapitel 13 bekannt und generiert den Programmkopf und erwartet als Parameter zwei Listen. Diese beiden Listen werden durch die CONSTITUENTS- Konstrukte berechnet. Laut Grammatik beinhaltet der Knoten `specification` ein oder mehrere Schemadefinitionen (`schema_def`) und ein oder mehrere Pfaddefinitionen (`path_def`), also Angaben aus dem Mappingblock. Von diesen vorkommenden Knoten werden jeweils die Ptg-Attribute selektiert. Da es in diesem Falle mehrere solcher Knoten geben kann, wird nicht das bislang bekannte `CONSTITUENT` verwendet, welches ausschließlich für das einfache Auftreten eines Knotens verwendet werden kann, sondern das CONSTITUENTS-Konstrukt. In diesem Falle müssen deshalb auch noch Angaben darüber gemacht werden was mit den einzelnen zurückgelieferten Attributen geschehen soll, bzw. wie diese Daten miteinander kombiniert werden sollen. Diese Angaben erfolgen durch das `WITH (Type, Combine, Convert, Create)`. *Type* spezifiziert hierbei den Typ, der letztendlich zurückgeliefert wird, während *Combine*, *Convert* und *Create* Funktionsaufrufe darstellen, die Ergebnisse dieses Typs liefern. *Combine* erhält dabei zwei Type-Objekte und liefert als Ergebnis ein *Type*-Objekt welches die beiden Objekte miteinander verbunden hat. In diesem Falle wird die Patternfunktion `PTGSeq` aufgerufen, welche einfach zwei PTG-Nodes hintereinander verknüpft. *Convert* erhält ein Objekt, welches durch den CONSTITUENTS-Ausdruck spezifiziert ist und wandelt dies in ein *Type*-Objekt um. Da in diesem Fall bereits durch den CONSTITUENTS-Ausdruck ein Objekt vom Typ `PTGNode` angegeben wurde, braucht keine Umwandlung stattfinden und die Funktion `IDENTICAL` liefert dasselbe Objekt wieder zurück. *Create* generiert ein entsprechendes Objekt. Die Funktion `PTGNull` ist durch das PTG-Modul vordefiniert und liefert einen `PTGNode`, der den leeren String repräsentiert. Alles in allem sorgt der CONSTITUENTS-Ausdruck also dafür alle definierten Ptg-Attribute, die im Syntaxbaum unterhalb des Knotens liegen, nacheinander zu selektieren und jeweils hintereinander zu hängen, sprich eine Ausgabensequenz zu generieren. Für den Fall des Symbols `specification` führt dies also zur Generierung des Programmkopfes und der Auflistung aller Schemadefinitionen gefolgt von der Auflistung aller Pfaddefinitionen. Diese PTG-Datenstruktur wird diesmal nicht in einem Ptg-Attribut abgelegt, sondern direkt an `PTGOut` weitergereicht und somit ausgegeben.

E.5.2 Definitionsblock

Im Definitionsblock werden nun für jede Schema-, Klassen-, Listen-, Pointer- und Attributdefinition entsprechende Ptg-Patterns ausgeführt und mit den entsprechenden Parametern versorgt. Da in der Grammatik die jeweiligen Symbole ähnlich derer Semantik benannt sind, ist die Spezifikation leichter nachvollziehbar.

```
SYMBOL schema_def COMPUTE
  SYNT.Ptg=
    PTGSchemaList(
      THIS.SchemaNamePtg,
      CONSTITUENTS class_decl.Ptg WITH (PTGNode, PTGSeq,
                                         IDENTICAL, PTGNull),
      THIS.SchemaTypePtg);
END;
```

Für jeden Knoten der Form `schema_def` wird ein neues Attribut `Ptg` generiert. Dieses beinhaltet als Wert die Rücklieferung der Funktion `PTGSchemaList`. Diese Funktion muß mit drei Parametern

aufgerufen werden. Der erste und der dritte Parameter beinhalten den Schemanamen und den Schematyp. Diese Ptg-Attribute wurden früher in diesem Kapitel bereits berechnet, bzw. in diesen Knoten hineinkopiert. Aus diesem Grunde kann nun sehr einfach darauf zugegriffen werden, wobei `THIS` den Knoten, auf dem momentan Berechnungen ausgeführt werden, spezifiziert - in diesem Falle also `schema_def`. Der zweite Parameter ist wiederum eine Liste, die weitere Symbolberechnungen notwendig macht.

```
SYMBOL class_decl COMPUTE
  SYNT.Ptg =
    PTGCoPoLi(
      CONSTITUENTS complex_decl.Ptg  WITH (PTGNode, PTGSeq,
                                           IDENTICAL, PTGNull),
      CONSTITUENTS pointer_decl.Ptg  WITH (PTGNode, PTGSeq,
                                           IDENTICAL, PTGNull),
      CONSTITUENTS compound_decl.Ptg WITH (PTGNode, PTGSeq,
                                           IDENTICAL, PTGNull)
      );
END;
```

In der Grammatik verzweigt das Symbol `class_decl` lediglich in die drei Möglichkeiten von Definitionen, die an dieser Stelle erlaubt sind. Dies spiegelt auch die obige Generierung des entsprechenden Ptg-Attributes wieder. Es werden lediglich drei verschiedene Listen mittels der Patternfunktion `PTGCoPoLi` aneinandergehängt. Im folgenden seien nicht mehr alle notwendigen Spezifikationen aufgeführt, da diese nach demselben Prinzip arbeiten. Es werden zur Veranschaulichung einige wichtige herausgegriffen.

```
SYMBOL simple_decl COMPUTE
  SYNT.Ptg=
    PTGAttributeList(
      CONSTITUENT simple_type.Ptg,
      CONSTITUENT attribute_name_def.Ptg,
      THIS.SchemaTypePtg,
      THIS.SchemaNamePtg,
      THIS.ComplexNamePtg);
END;
```

Hierbei handelt es sich um die Ausgabengenerierung für Attributdefinitionen. Für jeden Knoten `simple_decl` wird auch hier ein Ptg-Attribut generiert. Der Inhalt ist das Ergebnis des Funktionsaufrufes von `PTGAttributeList`. Aus der Grammatik wird deutlich, daß im Syntaxbaum unterhalb dieses Knotens noch Knoten mit den Angaben über Attributtyp und den Attributnamen vorliegen. Von diesen kommt jedoch jeweils genau einer vor, so daß auf deren Ptg-Attribute mittels des einfachen CONSTITUENT-Ausdruckes zugegriffen werden kann. Die Ptg-Attribute, die den Schematypen, Schemanamen und Klassennamen beinhalten, wurden bereits durch vorhergehende Berechnungen hierher propagiert. Die Spezifikationen für die Knoten `compound_decl`, `pointer_decl` und `generic_decl` lauten analog.

Die so durchgeführte Verschachtelung der PTG-Funktionsaufrufe führt so zu der kompletten Ausgabe, die für den Definitionsteil des Eingabetextes notwendig ist.

E.5.3 Mappingblock

Aufwendiger und größer werden nun die Spezifikationen, die für die Ausgaben im Mappingteil des Eingabetextes notwendig sind.

Um den Einstieg zu finden, wird an die erste Spezifikation des Knotens `specification` aus dem letzten Abschnitt angeknüft. Das für die Ausgaben des Mappingblockes zuständige Ptg-Attribut ist demnach `path_def.Ptg`, von denen mehrere existieren können, weshalb es auch mittels `CONSTITUENTS` in die Spezifikation von `specification.Ptg` eingebunden ist.

```
SYMBOL path_def COMPUTE
 SYNT.Ptg=
  PTGPathName(
   CONSTITUENT path_name_def.Ptg,
   CONSTITUENTS mapping_schema_from.Ptg WITH (PTGNode, PTGSeq,
                                             IDENTICAL, PTGNull),
   CONSTITUENTS mapping_schema_to.Ptg WITH (PTGNode, PTGSeq,
                                           IDENTICAL, PTGNull));
END;
```

Für die Ausgabe wird die Patternfunktion `PTGPathName` aufgerufen, die in Kapitel 13 spezifiziert wurde. Sie erhält drei Parameter, aus denen sie ihre Ausgaben generieren wird. In der Ausgabe erscheint zunächst ein Kommentar, der lediglich den Pfadnamen beinhaltet, um den Ausgabetext übersichtlicher zu strukturieren. Dafür ist die Übergabe von `path_name_def.Ptg` notwendig. Innerhalb dieser Pfaddefinitionen können, wie man aus der Grammatik ableiten kann, nur zwei Arten von Definitionen in der Eingabedatei erfolgen: entweder ein Mapping-To oder ein Mapping-From. Die Ausgaben, die dafür notwendig sind, werden in den entsprechenden Nachfolgeknoten berechnet und werden deshalb an dieser Stelle als zwei mögliche Listen übergeben. Diese Spezifikation spiegelt sehr deutlich die Parallelität zur Grammatik wieder. Innerhalb eines Pfades erscheint zwar immer nur jeweils eine der beiden Ausgaben, also entweder ein Mapping-To oder ein Mapping-From, aber das nebeneinander in der Spezifikation birgt dabei keine Schwierigkeiten, da für den jeweils anderen *leere* Ausgaben vom Typ `PTGNull` generiert werden.

```
SYMBOL mapping_schema_from COMPUTE
 SYNT.Ptg=
  PTGMapSchemaFrom(
   PTGAddPostSchema(
    CONSTITUENT schema_specifier_cs.SchemaTypePtg,
    CONSTITUENT schema_specifier_cs.SchemaNamePtg,
    CONSTITUENT schema_specifier_es.SchemaTypePtg,
    CONSTITUENT schema_specifier_es.SchemaNamePtg),
   PTGAddPostSchema(
    CONSTITUENT schema_specifier_es.SchemaTypePtg,
    CONSTITUENT schema_specifier_es.SchemaNamePtg,
    CONSTITUENT schema_specifier_fs.SchemaTypePtg,
    CONSTITUENT schema_specifier_fs.SchemaNamePtg),
    CONSTITUENTS mapping_type_from.Ptg WITH (PTGNode, PTGSeq,
                                            IDENTICAL, PTGNull));
END;
```

Die Spezifikationen, die nun folgen, ähneln sich im Aufbau. Die Ausgabe, die von mapping_schema_from generiert werden, bestehen hierbei aus dem Aufruf der Funktion PTGMapSchemaFrom. In diesem Fall werden als Übergabewerte weitere PTG-Funktionen aufgerufen, die zunächst die entsprechenden PTG-Strukturen generieren, bevor diese dann an PTGMapSchemaFrom übergeben werden. Dies erscheint an dieser Stelle sinnvoll, denn die Parameter für die Aufrufe von PTGAddPostSchema sind eindeutig und direkt zugreifbar, weshalb also keine CONSTITUENT-Konstrukte benötigt werden. Eine weitere Delegation der PTG-Berechnung in einen tiefer liegenden Knoten würde die Spezifikation deshalb an dieser Stelle nur unnötig verkomplizieren. Die Verschachtelung in die nächste Knotenebene erfolgt bei der Übergabe des dritten Parameters, welche wiederum in Form einer Liste erfolgt. Da innerhalb eines Mapping-From-Abschnittes auch in tiefer liegenden Knoten nur *From*-Kontrukte Verwendung finden, handelt es sich hierbei nur um eine lineare Verschachtelung. Es existieren also nicht wie bei der Spezifikation von path_def mehrere Möglichkeiten der Verzweigung, von denen jeweils nur eine in Frage kommt. Die Spezifikationen von map_type_from verlaufen in analoger Form. Erst bei dem Knoten map_from taucht wieder eine Verzweigung auf.

```
SYMBOL map_from COMPUTE
  SYNT.Ptg=
    PTGMapFrom(
      CONSTITUENTS map_from_simple.Ptg WITH (PTGNode, PTGSeq,
                                             IDENTICAL, PTGNull),
      CONSTITUENTS map_from_proc.Ptg   WITH (PTGNode, PTGSeq,
                                             IDENTICAL, PTGNull),
      CONSTITUENTS map_from_nest.Ptg   WITH (PTGNode, PTGSeq,
                                             IDENTICAL, PTGNull));
END;
```

In der nächst folgenden Knotenebene kommen nämlich nur noch drei verschiedene Möglichkeiten in Frage. Danach ist man im Syntaxbaum in den Blättern angelangt. Die Art der Funktionsaufrufe ist nun schon ausreichend bekannt. Für die Knoten map_from_simple und map_from_proc sind die Spezifikationen leicht abzuleiten und sind denen aus den vorhergehenden Abschnitten ähnlich. Analog zu den Spezifikationen für den Mapping-From Bereich gestalten sich auch die Angaben für den Mapping-To Bereich. Die Unterschiede liegen dabei ausschließlich bei den zu benutzenden Variablennamen.

Besonderheiten weisen nur diejenigen Knoten auf, die schon in Abschnitt 14.4.2 und 14.4.4 für komplexere Spezifikationen gesorgt haben. Die Ausgabengenerierung dieser Knoten erweist sich auch hier als schwieriger. Für den Fall map_from_nest handelt es sich nur um eine weitere Verschachtelung der Berechnung. Für jedes der map_select_multi_cs-Knoten müssen jeweils PTG-Ausgaben generiert werden, so daß sich dies nicht als größere Schwierigkeit herausstellt und die Spezifikationen dafür auch analog zu den bisher bekannten Schachtelungen gemacht werden können. Zu beachten ist hierbei nur, daß für die Knoten map_select_multi_cs bereits Ptg-Attribute berechnet wurden, die als Wert die Stringrepräsentation des entsprechenden Eingabetextes beinhalten. Da aber für diese Knoten auch eine eigene PTG-Ausgabe mittels Aufrufe einer entsprechenden Funktion generiert werden müssen, wird in diesem Falle ein Attribut mit dem Namen OutPtg neu definiert, welches die PTG-Struktur dann enthält. Da dies auch für die Knoten map_select_multi_is gilt, sei diese Spezifikation hier aufgeführt:

```
SYMBOL map_select_multi_is COMPUTE
  SYNT.OutPtg=
    PTGTripple(
      PTGAddPreAttribute(
        THIS.FSSchemaTypePtg,
        THIS.FSSchemaNamePtg,
        THIS.FSTypeNamePtg,
        THIS.PTGNodeElem,
        THIS.ISSchemaTypePtg,
        THIS.ISSchemaNamePtg,
        THIS.ISTypeNamePtg,
        THIS.Ptg),
      PTGAddPreAttribute(
        THIS.ISSchemaTypePtg,
        THIS.ISSchemaNamePtg,
        THIS.ISTypeNamePtg,
        THIS.Ptg,
        THIS.CSSchemaTypePtg,
        THIS.CSSchemaNamePtg,
        THIS.CSTypeNamePtg,
        THIS.CSAttributeNamePtg),
      PTGAddPostParameter(
        THIS.CSSchemaTypePtg,
        THIS.CSSchemaNamePtg,
        THIS.CSTypeNamePtg,
        THIS.CSAttributeNamePtg,
        THIS.ISSchemaTypePtg,
        THIS.ISSchemaNamePtg,
        THIS.ISTypeNamePtg,
        THIS.Ptg));
END;
```

Dies ist auch genau die Stelle, an der die LidoList aus Abschnitt E.3.3 Verwendung findet. Im ersten PTG-Funktionsaufruf `PTGAddPreAttribut` wird als vierter Parameter ein Listenelement übergeben. An dieser Stelle werden für jeden Knoten `map_select_multi_is` Ausgabeberechnungen durchgeführt, was mit Hinblick auf den Eingabetext als sequenzielle Abarbeitung der Attributliste unter *MAP-VIA* anzusehen ist. Für die Ausgabe wird jeweils, bezüglich seiner Position, das passende Attribut aus der Liste unter *MAP* benötigt. Die dafür benötigte Liste wurde bereits diskutiert und auch schon für die Benutzung vorbereitet, so daß ihre Verwendung innerhalb der Knotenberechnung nun kaum ins Auge fällt, bemessen an dem Aufwand der getätigt werden mußte bei der Konstruktion der Liste.

E.6 Die Spezifikationsdatei für die Ausgabe

Es handelt sich bei den Berechnungen für die Ausgabengenerierung auch um Symbolberechnungen. Diese werden in Eli in Dateien vom Typ `.lido` angegeben. Die gesammelten Spezifikationen aus Abschnitt 14.5 finden sich demnach in einer Datei mit dem Namen `ausgabe.lido` wieder. Da

die Spezifikationen in den Dateien `ausgabe.lido` und `ausgabe.ptg` (siehe Abschnitt D.4) von ihrer Semantik sehr nah zusammenliegen, wird an dieser Stelle davon Gebrauch gemacht diese in einer Datei mit dem Namen `ausgabe.fw` zusammenzufassen. Dies ist durch die Unterstützung des Literate-Programming durch Eli möglich. Die Spezifikationen aus der Datei `ausgabe.ptg` enthalten die Ausgabepatterns, die in den Spezifikationen aus der Datei ausgabe.lido Verwendung finden.

Anhang F

Erläuterungen zum Laufzeitsystem

F.1 Konfigurationsdatei für O_2Makegen

Im Folgenden ist beispielhaft eine Konfigurationsdatei für den Makefilegenrator von O_2 aufgeführt, die es möglich macht die Klassen des Föderierungsgraphen in O_2 zu verwenden.

```
; Angaben ueber die Verbindung zu O2
O2Home : /home/software/O2-2
O2System : FDBS
O2Server : athen
O2Schema : Diplomarbeit
; Optionen
+UseConfirmClasses

ImpFiles: graph.H
[graph.H]ImpClasses:  CConvert CSchema CComplexType CType\
CObject CGraph CVariable CAttribute CCollectionType CPointerType\
CSimpleType TAttribute TSAttribute CIdentifyConvert CConcatConvert\
CEuroToDmConvert CInsert

ImpList: CAttribute TAttribute TSAttribute CObject CGraph CSchema\
CType CVariable

; Sourcen
Sources: TAttribute.C TSAttribute.C CConvert.C CIdentifyConvert.C\
CConcatConvert.C CEuroToDmConvert.C CAttribute.C CCollectionType.C\
CComplexType.C CGraph.C CObject.C CPointerType.C CSchema.C\
CSimpleType.C CType.C CVariable.C CSpecialAlgorithms.C CInsert.C main.C

; zu erzeugende Objectdateien
ProgramObjs: TAttribute.o TSAttribute.o CConvert.o CIdentifyConvert.o\
CConcatConvert.o CEuroToDmConvert.o CAttribute.o CCollectionType.o\
CComplexType.o CGraph.o CObject.o CPointerType.o CSchema.o CSimpleType.o\
CType.o CVariable.o CSpecialAlgorithms.o CInsert.o main.o

; Programmname
ProgramName: main
```

Anhang G

Das Testsystem

G.1 Realisierung der Grapherzeugung

Um den konkreten Graphen in der Datenbank O_2 anzulegen, wird auf die Ausgabe des Compilers zurückgegriffen. Der vom Compiler erzeugte Code wird in eine Datei mit dem Namen `build.C` ausgegeben. In diesem Code werden die beiden Funktionen `BuildGraph` und `BuildSimpleTypes` definiert. Vom Compiler wird dafür vorgegeben, daß eine *C*++-Headerdatei mit dem Namen build.H existieren muß, welche die beiden Funktionen spezifiziert. Diese Headerdatei ist unabhängig von der Compilerausgabe und sieht wie folgt aus:

```
#include "graph.H"

// Funktionsprototypen
void BuildGraph( PCGraph pg );
void BuildSimpleTypes( PCSchema ps );
```

Die Datei `graph.H` ist durch das Laufzeitsystem in O_2 vorgegeben und beinhaltet alle notwendigen Klassendefinitionen und O_2-spezifischen Spezifikationen.

Um einen Graphen in der Datenbank O_2 anlegen zu können, müssen folgende Bedingungen erfüllt sein:

- Der O_2-Datenbankserver muß laufen. Dafür muß ein gültiges System eingetragen sein. In diesem Falle ist dies mit dem Namen `FDBS` versehen.
- Im O_2-System muß ein Schema existieren. In diesem Falle wurde es `Diplomarbeit` genannt.
- Sämtliche Klassenimplementationen für das Laufzeitsystem müssen vorliegen, falls ein Import der Graphklassen in die Datenbank zuvor noch nicht geschehen ist.
- Mit Hilfe eines Werkzeuges von O_2 muß ein Makefile angelegt sein, welches neben den Graphklassen auch die Dateien `build.C` und `buildgraph.C` (siehe Abschnitt G.1.1) übersetzt und den generierten, ausführbaren Code unter dem Namen `buildgraph` ablegt. Der Aufbau eines solchen Makefiles, bzw. die Benutzung der entsprechenden Werkzeuge wird in [O297] beschrieben.

Die Vorgehensweise zur Grapherzeugung kann nun wie folgt skizziert werden:

1. Öffnen des Datenbanksystems FDBS.
2. Eintragen einer neuen *Base* in das Schema Diplomarbeit. Bei einer *Base* handelt es sich um eine Datenbank innerhalb eines Schemas. Erst in eine solche Datenbank können Daten persistent abgelegt werden.
3. Eintragen einer sogenannten *persistent Root* in diese Base. Sie bildet den späteren Einstiegspunkt in die Datenbank für das Föderierungssystem. Sie wird in diesem Fall mit dem Namen Graph versehen.
4. Aus Sicherheitsgründen werden diese Datenbankoperationen jeweils in Transaktionen gebettet. Dies hat zur Folge, daß der Einstiegspunkt, die *persistent Root*, verloren geht und für das weitere Vorgehen neu aus der Datenbank geholt werden muß.
5. Der so neu gewonnene Zeiger auf den Graphen in der Datenbank wird an die Funktion BuildGraph, die vom Compiler generiert wurde, übergeben. Diese Funktion legt nun den konkreten Graphen persistent in der Datenbank an. Auch dieser Vorgang muß in eine Transaktion eingebettet werden.
6. Zum Abschluß wird die Datenbank ordnungsgemäß geschlossen.

Der für dieses Programm notwendige *C++*-Quellcode findet sich in Abschnitt G.1.1 und ist ausreichend kommentiert. Die Datei trägt den Namen buildgraph.C.

Eine Aufruf von make führt nun dazu, daß dieser Code im Zusammenhang mit O_2 übersetzt wird. Falls die Graphklassen zuvor noch nicht in das Datenbankschema importiert wurden, so wird auch dieser Vorgang zunächst ausgeführt. Das resultierende ausführbare Programm wird nach Programmstart den Graphen anlegen.

G.1.1 *C++*-Quellcode für die Graphgenerierung

```
// buildgraph.C

// O2-Includes

#include "o2lib_CC.hxx"
#include "o2template_CC.hxx"
#include "o2util_CC.hxx"

// Graph-Klassen-Include                                          10

#include "graph.H"

// Include der Buildroutine der Compilerausgabe
#include "build.H"

// MAIN

main(int argc, char *argv[])
{                                                                 20
  cout << "Buildgraph startet."<<endl<<flush;
```

```
// Datenbankvariablen
d_Session session;
d_Database database;
d_Database *podatabase = &database;
d_Transaction transaction;

// stelle Verbindung zur Datenbank her
if ( session.begin (argc,argv,"FDBS","athen","/home/software/O2-5") )      30
  {
    cerr << "Something wrong to start O2"<<endl;
    exit(1);
  }

// Transactionsbeginn fuer Erzeugung neuer Datenbank
transaction.begin();

// Erzeuge die Datenbank TESTGRAPH
database.create("TESTGRAPH","Diplomarbeit");      40

// Commit auf Datenbankgenerierung
transaction.commit();

// Beginne Transaction fuer Graphgenerierung
transaction.begin();

// Erzeuge eine Persistent-Root namens "Graph"
database.create_persistent_root("Graph","d_List<d_Ref<CGraph> >");
transaction.commit();      50

// Generiere nun in die Datenbank den Graph

// Erzeuge eine Graphvariable und verbinde sie mit der
// Persistent-Root
transaction.begin();

CGraph *theGraph;
theGraph=new(podatabase) CGraph;
      60
// Hole Persistent-Root-Liste
d_List<PCGraph> rootList("Graph");
rootList.remove_all();

// Fuege Graphvariable in die Liste ein, um diese persistent zu machen

rootList.insert_element_last(theGraph);
transaction.commit();

// hole Persitent-Root      70
transaction.begin();
d_List<PCGraph> RootList1("Graph");

//  Hole erstes Element aus der Liste -> Der Graph!
PCGraph TheGraph1 = new CGraph;
TheGraph1 = RootList1.retrieve_first_element();

// rufe Buildgraph auf. Diese Methode ist in der Compilerausgabe definiert
// und wird hier durch das include auf build.H eingebunden
BuildGraph(TheGraph1);      80
```

```
    // Commit auf die Graphgenerieung
    transaction.commit();

    // Schliesse Datenbank
    database.close();
    session.end();
}
```

G.2 Realisierung der Konvertierungsfuktionen

Für das Testsystem müssen die Konvertierungsfunktionen `Concat` und `EuroToDm` implementiert werden. Als Grundlage dazu dient die für den Graphen definierte Klasse `CConvert`. Bei dieser Klasse handelt es sich um eine sogenannte *abstrakte* Klasse. In ihr ist die allgemeine Schnittstelle definiert, die für alle Konvertierungsfunktionen zu benutzen ist. Die Klasse enthält als einziges die Methode `Convert`. Der Methodenname steht eindeutig fest, so daß die Graphalgorithmen, die später auf dem Graphen arbeiten, für jedes Attribut eine eindeutige Konvertierungsfunktion vorfinden, die sich nur inhaltlich unterscheiden können. Um eine neue Konvertierungsmethode zu implementieren, wird also eine neue Klasse geschaffen, welche die Schnittstelle von der Klasse `CConvert` erbt. Der Name der neuen Klasse ist durch die Spezifikation vorgegeben. Er beginnt mit einem 'C' und endet auf 'Convert'. Dazwischen ist der Name der Konvertierung aus der Spezifikation zu verwenden. Es ist hierbei auf Groß-/Kleinschreibung zu achten.

Die Schnittstelle der Klasse `CConvert` ist dabei so allgemein gehalten, daß die später implementierte Methode `Convert` mit allen notwendigen Informationen versorgt wird, bzw. sich die Informationen selbst beschaffen kann.

```
d_String CConvert::Convert ( PCAttribute poAttribute,
         TAttributeList poAttributeList, d_String oValue )
```

Die Methode `Convert` bekommt als Parameter den Zeiger auf das Attributobjekt im Föderierungsgraphen, in welchem sie selbst eingetragen ist. Dies ist notwendig, da das Attribut über die erforderlichen Informationen verfügt, aus welchen Vorgängerattributen es sich eventuell bei einem Nesting zusammensetzt. Der zweite Parameter ist eine Attribut-Wert-Paar-Liste, in der jedes Attribut und dessen Wert des zu konvertierenden Datensatzes enthalten ist. Als letzter Parameter wird ein String übergeben, der den Wert des aktuellen Attributes vor der Konvertierung enthält. Dieser Wert ist zwar auch in der vorher genannten Attribut-Wert-Liste enthalten, doch wenn lediglich eine Wertekonvertierung vorgenommen werden soll, also kein Nesting, so verfügt das Attribut im Föderierungsgraphen nicht über die Information des Vorgängerattributes, weil die Kanten im Graphen unidirektional gerichtet sind. Nur bei einem Nesting werden zusätzlich die Verbindungen zum Vorgänger Attribut eingetragen. Im Falle einer Wertekonvertierung muß also mit dem als letzten Parameter übergebenen String gerechnet werden. Als Rückgabewert liefert die Methode `Convert` in allen Fällen den Wert des Attributes nach der Konvertierung als String.

Für diesen Anwendungsfall müssen also die Klassen `CConcatConvert` und `CEuroToDmConvert` geschaffen werden und jeweils die Methode `Convert` innerhalb dieser Klassen implementiert werden.

Die bei der Implementierung in *C++* notwendigen Klassendeklarationen werden in die Datei des Laufzeitsystems `graph.H`, in der auch die Klassen `CConvert` und `CIdentifyConvert` deklariert sind, eingetragen. Dazu sind lediglich die Deklarationen aus der Klasse `CConvert` zu kopieren und der Name der Klasse entsprechend abzuändern. Die Implementierung der Klasse findet dann in einer eigenen *C++*-Datei statt, welche die Deklarationsdatei `graph.H` einbindet.

G.2.1 Die Klasse CConcatConvert

Die Methode `Convert` dieser Klasse holt sich aus dem Föderierungsgraphen die Zeiger der Attribute, aus deren Inhalt sie den Wert des neuen Attributes zusammensetzen soll (concat). Dies geschieht über die Funktion `GetPostParameterList()`. Diese Liste wird durchlaufen und jedes der darin enthaltenen Attribute wird in der übergebenen Attribut-Wert-Liste gesucht und dessen Inhalt aneinander gehängt. Der somit neu gewonnene Wert des Attributes nach der Konvertierung wird anschließend zurückgeliefert.

Der dafür notwendige *C++*-Quellcode ist im Folgenden abgebildet.

```
//   CConcatConvert.C

#include "graph.H"

// Leerer Konstruktor
CConcatConvert::CConcatConvert() {};

// Convert
                                                                                              10
d_String CConcatConvert::Convert ( PCAttribute poAttribute, TAttributeList poAttributeList, d_String oValue )
{
  oValue = "";
  d_List<PCAttribute> PostParameterListe;
  PostParameterListe=poAttribute->GetPostParameterList();
  d_Iterator<PCAttribute> i = PostParameterListe.create_iterator();
  PCAttribute currentAttribute;
  d_Ref<TAttribute> currentStruct;
  d_Iterator<d_Ref<TAttribute> > i_AList = poAttributeList.create_iterator();
                                                                                              20
  while (i.next(currentAttribute))
    {
      i_AList.reset();
      while (i_AList.next(currentStruct))
        if (currentAttribute == currentStruct->Attribute)
          oValue += currentStruct->Value;
    }
  return oValue;
}
```

G.2.2 Die Klasse CEuroToDmConvert

Die Methode `Convert` dieser Klasse bedient sich ausschließlich des letzten Parameters, der ihr übergeben wird, denn es handelt sich bei dieser Klasse lediglich um eine Wertekonvertierung.

Der String `oValue` enthält einen Zahlenwert in der Währungseinheit Euro. Für eine Konvertierung zur Deutschen Mark wird dieser Wert der Einfachheit halber mit 2 multipliziert. Der String muß

also zuvor in eine Zahl umgewandelt werden und nach der Umrechnung wieder in eine Zeichenkette zurückgeschrieben werden.

Der für diese Konvertierung notwendige *C++*-Quellcode ist leicht verständlich und ist im Folgenden aufgeführt.

```
// CEuroToDmConvert.C

#include "graph.H"

// Leerer Konstruktor
CEuroToDmConvert::CEuroToDmConvert()
{};

                                                                                  10
d_String CEuroToDmConvert::Convert ( PCAttribute poAttribute, TAttributeList poAttributeList, d_String oValue )
{
  float wert;
  wert = atof (oValue);
  wert = wert * 2.0;

  char* s;
  s = (char*) malloc(100);

  sprintf (s, "%f", wert);                                                        20
  oValue = s;
  return oValue;
}
```

G.3 Die Graphalgorithmen

Die Aufgabe der Graphalgorithmen ist es, die eigentliche Föderation der neu eingefügten Datensätze zu übernehmen. Für diesen Anwendungsfall wird lediglich die Funktion *Insert* unterstützt, da nur neue Datensätze eingefügt werden, um zu zeigen, daß die Wertekonvertierung und das Nesting funktionieren. Die Operationen *Update* und *Delete* werden deshalb nicht weiter betrachtet. Für die Implementierung wird eine Klasse `CSpecialAlgorithms` deklariert, welche die notwendigen Methoden zusammenfaßt. Die Namensgebung geht auf [Pro97] zurück, da auch dort schon umfangreiche Graphalgorithmen implementiert wurden.

Die Klasse `CSpecialAlgorithms` enthält demnach als von außen zugängliche Methode nur `Insert`, die speziell auf das Testsystem abgestimmt wurde. Sie erhält als Parameter eine Attribut-Wert-Paar-Liste. Diese enthält von dem zu föderierenden Datensatz alle Attribute in Form von Zeigern und die dazugehörenden Inhalte als Zeichenkette. Zurückgeliefert wird ein Zeiger auf ein Objekt der Klasse `CInsert`. Dieses Objekt beinhaltet ebenfalls eine Attribut-Wert-Liste, allerdings sind diesmal auch die Attribute mit ihrem Namen als Zeichenkette verwendet worden, damit diese Daten vom Föderierungskern sofort verschickt werden können. Darüber hinaus enthält das Ergebnisobjekt auch noch den Schema- und Klassenamen des zu föderierenden Datensatzes. Aus diesen Informationen könnte der Föderierungskern bei mehreren angeschlossenen Datenbanken auf die Datenbank schließen, in die der Datensatz einzufügen ist.

Die für *C++*-Implementierungen notwendige Deklarationsdatei, in diesem Falle `CSpecialAlgorithms.H` sieht demnach wie folgt aus:

```
#include "graph.H"

class CSpecialAlgorithms
{
public:
  // leerer Konstruktor
  CSpecialAlgorithms();

  CInsert* Insert(TAttributeList poAttributeList);

private:
  TAttributeList ConverttoES (TAttributeList paOrgList);
  TAttributeList ConverttoIS (TAttributeList paESList);
  TAttributeList ConverttoCS (TAttributeList paISList);
  CInsert* createCInsert (TAttributeList paCSList);
  bool HasAttribute(TAttributeList suchList,
                    d_Ref<TAttribute> attribute);
};
```

Da der Föderierungsgraph aus [Pro97] um die Funktionalität des Nesting und der Wertekonvertierung erweitert wurde, ist nun auch eine völlig neue Strategie für die Methode `Insert` notwendig. Sie läßt sich durch die folgenden Punkte skizzieren:

1. Der übergebene Datensatz wird gemäß der Spezifikation zum Exportschema konvertiert.

2. Der resultierende Datensatz wird gemäß der Spezifikation zum Importschema konvertiert. Da bei dieser Überführung keine Wertekonvertierungen und Nestings definiert sind, kann auf eine Konvertierung ins föderierte Schema verzichtet werden. Es werden dort direkt die Informationen zum Importschema gesammelt.

3. Der resultierende Datensatz wird gemäß der Spezifikation zum Komponentenschema konvertiert.

4. Der so erhaltene Datensatz wird in ein Objekt der Klasse `CInsert` konvertiert und zurückgegeben.

Die einzelnen Konvertierungen zwischen den Schemata werden in getrennte Methoden gekapselt, sodaß die Methode `Insert` sehr kurz und übersichtlich wird. Der Quellcode hierfür befindet sich u.a. in Abschnitt G.3.1.

Die Vorgehensweise bei den drei Konvertierungen zwischen den Schemata läßt sich durch die folgenden Punkte beschreiben:

1. Für ein Attribut des Datensatzes wird das Vorgänger-, bzw. Nachfolgerattribut ermittelt. Der so ermittelte Zeiger ist ein Teil des neuen Attribut-Wert-Paares.

2. Der Wert des neuen Attributes wird ermittelt, indem im zuvor ermittelten Attribut die Konvertierungsmethode `Convert` mit den erforderlichen Parametern aufgerufen wird. Diese Parameter sind zu diesem Zeitpunkt alle bekannt und müssen nicht aufwendig ermittelt werden.

3. Das so neu gewonnene Attribut-Wert-Paar wird in die Rückgabeliste der Methode eingetragen, falls es noch nicht zuvor eingetragen wurde. Diese Überprüfung ist auf Grund des Nestings notwendig. Die dafür notwendige Methode `Has` ist ebenfalls in der Klasse `CSpecialAlgorithms` implementiert.

4. Der Vorgang wird für jedes Attribut des zu föderierenden Datensatzes wiederholt.

Da sich die drei notwendigen Konvertierungen sehr ähnlich sind, sei in Abschnitt G.3.1 als Beispiel der *C++*-Quellcode für `ConvertToCS` aufgeführt.

Bei der Konvertierung des so erhaltenen Datensatzes in ein `CInsert`-Objekt finden ausschließlich 1:1-Abbildungen statt, so daß diese einfache Methode hier nicht weiter beschrieben werden soll.

G.3.1 Die Methoden `Insert` und `ConvertToCS` der Klasse CSpecialalgorithms

```
// Insert
CInsert* CSpecialAlgorithms::Insert(TAttributeList paAttributeList)
{
  // Variablen und Listen
  CInsert* oInsert = new CInsert;
  TAttributeList paESAttributeList;
  TAttributeList paISAttributeList;
  TAttributeList paCSAttributeList;

  // konvertiere die Attributliste zum ES                                10
  paESAttributeList = ConverttoES (paAttributeList);

  // konvertiere die ES-Attributliste zum IS
  paISAttributeList = ConverttoIS (paESAttributeList);

  // konvertiere die IS-Attributliste zum CS
  paCSAttributeList = ConverttoCS (paISAttributeList);

  // generiere aus der CS-Attributliste ein CInsert-Objekt
  oInsert = createCInsert (paCSAttributeList);                         20

  return oInsert;
};

//  ConverttoCS
TAttributeList CSpecialAlgorithms::ConverttoCS( TAttributeList paISList)
{
  // lokale Variablen
  TAttributeList CSList; // Liste, die zurueckgeliefert wird
  PCAttribute CSAttribute; // temp.Attribut                            30
  d_List<PCAttribute> CSAttributeList; // Liste der Pre-Attribute

  // Iterator fuer die uebergebene Liste
  d_Iterator<d_Ref<TAttribute> > iISList = paISList.create_iterator();
  d_Ref<TAttribute> ISAttribute; // Element der uebergebenen Liste

  // fuer jedes Element der Liste wird das Pre-Attribut ermittelt
  while ( iISList.next(ISAttribute))
    {
      // Liste der Pre-Attribute                                       40
      CSAttributeList = (ISAttribute->Attribute)->Pre();
```

```
    // da die Liste nur ein Element enthaelt, wird dieses sofort entnommen
    CSAttribute = CSAttributeList.retrieve_first_element();

    // das CSTAttribute, welches in die Rueckgabeliste gehangen wird,
    // wird mit Wert gefuellt
    // konvertiertes Attribut-Wert-Paar
    d_Ref<TAttribute> CSTAttribute = new TAttribute;
    CSTAttribute->Attribute = CSAttribute;                                    50
    CSTAttribute->Value = CSAttribute->Convert( CSAttribute, paISList,
                                                ISAttribute->Value);

    // danach wird es in die Liste gehangen
    if (!HasAttribute(CSList,CSTAttribute))
      CSList.insert_element_last(CSTAttribute);
  }
  return CSList;
};
```

G.4 Der Föderierungskern

Der Föderierungskern bildet die Steuerungskomponente des FDBS. Er empfängt Daten von einer lokalen Datenbank, übergibt diese den Graphalgorithmen und sendet deren Ergebnis an eine weitere lokale Datenbank.

Der Aufbau des Kerns ist recht einfach. Zunächst wird die O_2-Datenbank, die den Föderierungsgraphen enthält, geöffnet. Anschließend wird die Hauptroutine des Föderierungskerns, die Funktion `handle`, aufgerufen. Hier werden in einer Endlosschleife Datensätze von einer lokalen Datenbank über eine *Unix-Pipe* empfangen und in eine Listenstruktur eingetragen. Diese Liste wird den Graphalgorithmen als Argument übergeben. Als Ergebnis wird wiederum eine Liste zurückgeliefert, die den konvertierten Datensatz als String-Repräsentation enthält. Der Datensatz wird danach über eine Pipe an den Agenten der Ziel-Datenbank geschickt.

G.4.1 Quellcode des Föderierungskerns

In nachstehendem Auszug aus dem Quellcode des Föderierungskerns ist die oben erwähnte Funktion `handle` aufgeführt.

```
// Foederierungskern

// Pipes
#define RECEIVE "/home/diplom/gerding/.outpipe"
#define SEND    "/home/diplom/gerding/.inpipe"

void handle(PCGraph pi_graph)
{
  TAttributeList      po_list;                                             10
  TSAttributeList     po_retlist;
  d_Ref<TSAttribute>  o_current;
  CInsert*            po_ret;
  PCSchema            po_schema;
  PCType              po_class;
  CSpecialAlgorithms  oSpecial;
  char                sInput[80];
```

```
  d_String        sAttr[20];
  d_String        sValue[20];
  int             count;

  ifstream recpipe(RECEIVE);
  if (!recpipe)
  {
    cerr << "Unable to open " << RECEIVE << endl;
    exit(0);
  }
  ofstream sendpipe(SEND,ios::out|ios::nocreate);
  if (!sendpipe)
  {
    cerr << "Unable to open " << SEND << endl;
    exit(0);
  }

  po_schema = pi_graph->GetSchema("CSbook_1");
  po_class = po_schema->GetType("Buch");

  count = 0;
  while (recpipe.eof()) {}
  while (TRUE)
  {
    recpipe.getline(sInput, 80);
    if (strcmp(sInput, "#EOO#") == 0)
    {
      po_ret = oSpecial.Insert(po_list);
      po_retlist = po_ret->m_aAttributes;
      d_Iterator<d_Ref<TSAttribute> > i = po_retlist.create_iterator();
      while (i.next(o_current))
      {
        sendpipe << o_current->sValue << endl;
      }
      sendpipe << "#EOO#" << endl;
      sendpipe.flush();
      po_list.remove_all();
      count = 0;
    }
    else
    {
      sAttr[count] = sInput;
      recpipe.getline(sInput, 80);
      sValue[count] = sInput;
      po_list.insert_element_last(make_tupel(po_class, sAttr[count], sValue[count]));
      count++;
    }
  }
}
```

G.5 Agenten der Komponentendatenbanken

Für die Anbindung der lokalen Datenbanken an das Föderierungssystem sind zwei Agenten notwendig. Ihre Funktion ist das Versenden und Empfangen von Datensätzen. Der Agent zum Versenden von Datensätzen an das Föderierungssystem ist in die Datenbankapplikation integriert. Diese Kombination ist notwendig, da *mySQL* nicht über Trigger verfügt, mit denen eine Übertragung von Datensätzen an das Föderierungssystem unabhängig von einer DB-Applikation möglich wäre. Nach dem erfolgreichen Eintragen eines Datensatzes in die Datenbank wird selbiger als eine Folge von Attributnamen und zugehörigen Werten in eine *Pipe* geschrieben. Die *Pipe* realisiert dabei eine *FIFO*-Datenstruktur (*First in – First out*) auf Dateibasis, d.h. ein Prozeß kann Daten in eine *Pipe* schreiben, während ein anderer die Daten in Einfügereihenfolge aus der *Pipe* auslesen kann.

Die Applikation `bookdb` selbst verfügt über eine einfache, textbasierte Benutzungsoberfläche. Vom Hauptmenü aus sind drei Funktionen abrufbar. Die Appliaktion wird mittels `bookdb <dbname>` gestartet.

Datensatz eingeben: Über eine Eingabemaske lassen sich alle für einen Datensatz notwendigen Informationen erfassen. Nach der Bestätigung der Eingabe wird der Datensatz sofort in die Datenbank eingetragen, sowie an das Föderierungssystem übergeben.

Daten ausgeben: Alle Datensätze der Tabelle `Buch` der Datenbank werden ausgelesen und angezeigt.

Programm beenden: Die Datenbank wird geschlossen und das Programm beendet.

Der Agent zum Empfangen von Datensätzen vom Föderierungssystem ist einfach aufgebaut. Nach dem Öffnen der lokalen Datenbank werden in einer Endlosschleife Daten aus der *Pipe* gelesen, in die das FDBS zuvor Daten schreibt. Die empfangenen Strings werden zu einem Datensatz zusammengefügt und in die Datenbank geschrieben. Anschließend werden die nächsten Daten aus der *Pipe* gelesen, usw.

Der Agent wird mittels `inagent <dbname>` gestartet.

G.5.1 bookdb

Auszüge aus der Datei `bookdb.c`. Es sind nur die Funktionen zum Eintragen eines Datensatzes in die Datenbank und zur Ausgabe des Inhalts der Datenbank aufgeführt.

```
/*----------------------------------------------------------*/
/*   bookdb.c  -  Datenbankapplikation zum Test des FDBS
/*----------------------------------------------------------*/

#define INSERT_CMD "INSERT INTO Buch VALUES ('%s', '%s', '%s', '%s', '%f')"
#define SELECT_CMD "SELECT * FROM Buch"
#define OUTPIPE    "/home/diplom/gerding/.outpipe"

typedef struct {                                                    10
  char *titel;
  char *verfasser;
  char *standort;
  char *buchnr;
  char *signatur;
```

```
  float preis;
  } book;

MYSQL      mysql, *sock;
FILE       *fp;                                                    20
char       qbuf[300];

/*-----------------------------------------------*/
/* Datensatz in DB eintragen
/*-----------------------------------------------*/
void insert_entry()
{
  book* mybook;

  mybook = insert_mask(); /* Datensatz eingeben */                  30
  mask_clear();

  if (mybook != NULL)
  {
    sprintf(qbuf, INSERT_CMD, mybook->titel, mybook->verfasser,
            mybook->standort, mybook->buchnr, mybook->preis);
    if(mysql_query(sock, qbuf) < 0)
    {
      mvprintw(10, 5, "Query failed (%s)\n",mysql_error(sock));
      mvprintw(15, 5, "Taste drücken!");                            40
      refresh();
      gl_waitkey();
    }
    else
    {
      fprintf(fp, "%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%f\n#EOO#\n",
              "Titel", mybook->titel,
              "Verfasser", mybook->verfasser,
              "Standort", mybook->standort,
              "Buchnr", mybook->buchnr,                             50
              "Preis", mybook->preis);
      fflush(fp);
      mvprintw(10, 30, "Datensatz eingefügt!");
      refresh();
      sleep(1);
    }
  }
}

/*-----------------------------------------------*/                60
/* Inhalt der DB ausgeben
/*-----------------------------------------------*/
void show_contents()
{
  MYSQL_ROW row;
  MYSQL_RES *result;
  int       line = 5;
  char      empty_str[] = "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0";
  char      *outstr;
                                                                   70
  mask_clear();
  sprintf(qbuf,SELECT_CMD);
  if(mysql_query(sock, qbuf) < 0 || !(result = mysql_store_result(sock)))
  {
```

```
    move(10, 5);
    printw("Query failed (%s)\n",mysql_error(sock));
    move(15, 5);
    printw("Taste drücken!");
    refresh();
    gl_waitkey();                                                                 80
  }

  outstr = (char *) malloc(30);
  contents_mask();        /* Bildschirmmaske füe die Ausgabe von Datensätzen */
  while ((row = mysql_fetch_row(result)))
  {
    memcpy(outstr, empty_str, 25); strncpy(outstr, row[0], 20);
    mvprintw(line, 2, outstr);
    memcpy(outstr, empty_str, 25); strncpy(outstr, row[1], 20);
    mvprintw(line, 24, outstr);                                                   90
    memcpy(outstr, empty_str, 25); strncpy(outstr, row[2], 10);
    mvprintw(line, 46, outstr);
    memcpy(outstr, empty_str, 25); strncpy(outstr, row[3], 10);
    mvprintw(line, 58, outstr);
    memcpy(outstr, empty_str, 25); strncpy(outstr, row[4], 6);
    mvprintw(line, 70, outstr);
    line++;
    if (line > 20)
    {
      mvprintw(22, 10, "Taste drücken!");                                         100
      gl_waitkey();
      contents_mask();   /* Bildschirmmaske füe die Ausgabe von Datensätzen */
      line = 5;
    }
  }
  mysql_free_result(result);
  free(outstr);
  mvprintw(22, 10, "Taste drücken!");
  gl_waitkey();
}                                                                                 110
```

G.5.2 inagent

Auszüge aus der Datei `inagent.c`. Es ist nur die Funktion aufgeführt, die Daten aus der Pipe liest und in die Datenbak einfügt.

```
/*-------------------------------------------------------------*/
/* Datenbanadapter zum Einfuegen von Elementen
/*-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <mysql.h>

#define INSERT_CMD "INSERT INTO BOOK VALUES ('%s', '%s', '%s', '%s')"            10
#define INPIPE "/home/diplom/gerding/.inpipe"

MYSQL      mysql, *sock;
char       qbuf[300];

/*-------------------------------------------------------------*/
/* Datensatz aus Pipe lesen und in Datenbank eintragen
```

```
/*------------------------------------------------------------*/
void insert_entries()
{                                                                      20
  FILE *fp;
  int end_of_object;
  int count;
  char buffer[80];
  char entry[4][80];

  if ((fp = fopen(INPIPE,"r")) == NULL)
  {
    fprintf(stderr,"Couldn't open pipe %s", INPIPE);
    exit(EXIT_FAILURE);                                                30
  }
  while (1)
  {
    while (feof(fp)) {}
    end_of_object = 0;
    count = 0;
    while (end_of_object == 0)
    {
      fgets(buffer, 80, fp);
      if (strncmp(buffer,"#EOO#",4) != 0)                              40
      {
        buffer[strlen(buffer) - 1] = '\0';
        strcpy(entry[count], buffer);
        count++;
      }
      else
      {
        printf("Inserting new entry\n");
        end_of_object = 1;
        sprintf(qbuf, INSERT_CMD, entry[0], entry[1], entry[2], entry[3], entry[4]);   50
        if(mysql_query(sock, qbuf) < 0)
          fprintf(stderr, "Query failed (%s)\n",mysql_error(sock));
      }
    }
  }
}
```

Literaturverzeichnis

[alg63] Revised Report on the Algorithmic Language ALGOL 60. *Communication of the ACM 6*, Seiten 1–17, Januar 1963.

[Cat96] R. Cattell. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufman, 1996.

[Cho56] N. Chomsky. Three Models for the Description of Language. *IRE Transaction on Information Theory IT-2*, Seiten 113–124, 1956.

[Cle88] J.C. Cleaveland. Building Application Generators. *IEEE Software 5*, Seiten 25–33, July 1988.

[Dat] T.c.X. DataKonsultAB. mySQL. http://www.tcx.se.

[Dob96] E.-E. Doberkat. A Language for Specifying Hyperdocuments. *Software - Concepts and Tools*, 17:163–172, April 1996.

[Ger99] Sven Gerding. Ein interaktiver Editor für eine Spezifikationssprache für föderierte Datenbankschemata. Diplomarbeit, Lehrstuhl 10, FB Informatik, Universität Dortmund, 1999.

[GHL+92] R.W. Gray, V.P. Henring, S.P. Levi, A.M. Sloane, und W.M. Waite. Eli: A Complete, Felxible Compiler Construction System. *Communications of the ACM35*, Seiten 121–131, Feb. 1992.

[Has97] W. Hasselbring. Federated Integration of Replicated Information within Hospitals. *International Journal on Digital Libraries*, 1(3):192–208, November 1997.

[HKN85] E. Horowitz, A. Kemper, und B. Narasimham. A Survey of Application Generators. *IEEE Software 2*, Seiten 25–33, Jan. 1985.

[Joh] S.C. Johnson. Yacc: Yet Another Compiler-Compiler. In *[KR78]*.

[Jos96] N. Josuttis. Schablonage, Die Standard Template Library. *iX*, (6):104–110, 1996.

[Kas94] U. Kastens. Construction of Application Generator Using Eli. Technischer Bericht: Reihe Informatik tr-ri-94-143, Universität-GH Paderborn, März 1994.

[Kas96] U. Kastens. Construction of Application Generator Using Eli. In Jan Bosch und Görel Hedin, Hrsg., *Workshop on Compiler Techniques for Application Domain Languages and Extensible Language Models*, Nummer 26, Seiten 30–36, Lund University, Sweden, April 1996. Department of Computer Science.

[Knu92] Donald E. Knuth. Literate Programming. Lecture Notes Number 27, CLSI (Center for the Study of Language and Information), Leland Stanford Junior University, 1992.

[Knu93] D.E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, Addison-Wesley Publishing Company, New York, 1993.

[KR78] B.W. Kernighan und D.M. Ritchie. *UNIX Programmer's Manual*. Bell Laboratories, seventh Auflage, 1978.

[Kru92] H. Krueger. Software Reuse. *ACM Computing Surveys 24*, Seiten 131–183, June 1992.

[LS] M.E. Lesk und E. Schmidt. Lex: A Lexical Analyzer Generator. In *[KR78]*.

[McC72] R.M. McClure. An Appraisal of Compiler Technology. In *Springs Joint Computer Conference*, Band 40, Montreal, N.J., 1972. AFIPS Conference Proceedings, AFIPS Press.

[O296a] O2. *O2 Documentation*. O2 Technology, 1996.

[O296b] O2. *ODMG C++ Binding Guide*. O2 Technology, 1996.

[O297] O2. *O2Makegen User Manual*. O2 Technology, 1997.

[Pad97] P. Padawitz. *Programmiersprachen und ihre Übersetzer*, Skript zur gleichnamigen Vorlesung, Abschnitt 1.1. , Lehrstuhl V, FB Informatik, Universität Dortmund, Germany, 1997.

[Pro97] Projektgruppe 290. FOKIS , Föderiertes objektorientiertes Krankenhausinformationssystem. Abschlußbericht, Lehrstuhl X, Universität Dortmund, Germany, 1997.

[Rut52] H. Rutishauer. Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen. *Mitteilungen aus dem Institut für angewandte Mathematik der ETH-Zürich*, 1952.

[SK96] Günter Sauter und Wolfgang Käfer. BRIITY - A Mapping Language Bridging Heterogeneity. Technischer Bericht, Daimler Benz AG, Research & Technology, Dept. CIM Research (F3P), 1996.

[SL90] A. Sheth und J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SWZ95] A. Schür, A. Winter, und A. Zündorf. Spezifikation und Prototyping graphbasierter Systeme. *Proceedings Fachtagung Software–Technik 1995, TU Braunschweig*, Seiten 86–97, Okt. 1995.

[Wai] W.M. Waite. A Complete Specification of a Simple Compiler. http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli/examples/eli_pascalE.html.

[Wai94] W.M. Waite, Hrsg. *Beyond LEX and YACC: How to Generate the Whole Compiler*. USENIX Winter Technical Conference, 1994. Invited Paper.

[WHK88] W. Waite, V.P. Henring, und U. Kastens. Configuration Control in Compiler Construction. *International Workshop on Software Version and Configuration Control '88*, 1988.