

Eberhard-Karls-Universität Tübingen  
Wilhelm-Schickard-Institut für Informatik

**Diplomarbeit**

**Umsetzung von OQL-Anfragen  
an relationale Datenbanken unter  
Verwendung eines Persistenzframeworks**

von  
Oriel Maute

*Danksagung:*

An dieser Stelle möchte ich mich bei allen bedanken, die mir bei der Erstellung dieser Arbeit hilfreich zur Seite standen. Besonders bedanke ich mich bei Herrn Prof. Dr. Klaeren für die Ausgabe des Themas und für die Betreuung der Arbeit. Der Firma IBL Letters GmbH danke ich für die Möglichkeit, dieses praxisnahe Thema bearbeiten zu können. Im Speziellen bin ich den Herren Dr. Gerhard Wanner und Klaus-Dieter Jäger von der Firma IBL für ihre Unterstützung bei Fragen über das Produkt POLAR® dankbar.

*Eigenständigkeitserklärung:*

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbst verfaßt habe und keine anderen als die aufgeführten Quellen und Hilfsmittel verwendet habe.

Tübingen, den 06. Dezember 1999

Oriel Maute  
Aichelbergweg 1a  
88239 Wangen i.A.

Erstkorrektor:  
Zweitkorrektor:  
Betreuer bei der Firma IBL:

Prof. Dr. Klaeren  
Prof. Dr. Güntzer  
Dr. Wanner

Version vom 8. Dezember 1999

# Inhaltsverzeichnis

<b>INHALTSVERZEICHNIS .....</b>	<b>4</b>
<b>1. EINLEITUNG.....</b>	<b>5</b>
<b>2. DER STANDARD ODMG 2.0 .....</b>	<b>9</b>
2.1. OBJECT MODEL .....	10
2.2. OBJECT SPECIFICATION LANGUAGES.....	11
2.3. OQL.....	13
2.4. DAS JAVA-LANGUAGE BINDING.....	46
<b>3. SQL.....</b>	<b>47</b>
3.1. GESCHICHTLICHE ENTWICKLUNG VON SQL.....	47
3.2. TEILMENGE VON SQL.....	48
<b>4. POLAR.....</b>	<b>54</b>
4.1. SYSTEMARCHITEKTUR.....	54
4.2. ABBILDUNG DES OO-DATENMODELLS IN DEM RDBS.....	56
4.3. SCHEMA - AUFBAU.....	60
4.4. DER POLAR-KERN .....	61
<b>5. GRUNDLAGEN DES COMPILERBAUS .....</b>	<b>64</b>
5.1. THEORETISCHE GRUNDLAGEN.....	64
5.2. DAS WERKZEUG JAVACC.....	68
5.3. DAS WERKZEUG JJTREE.....	71
<b>6. OQL - PARSER.....</b>	<b>74</b>
6.1. SCHLÜSSELWÖRTER.....	74
6.2. PRODUKTIONEN.....	75
6.3. SPEZIELLE JAVACC EINSTELLUNGEN .....	78
<b>7. UMSETZUNG VON OQL-ANFRAGEN .....</b>	<b>79</b>
7.1. PROBLEME .....	79
7.2. EINFACHE ÜBERLEGUNGEN .....	82
7.3. GROBER ENTWURF .....	88
7.4. DETAILLIERTER ENTWURF.....	89
7.5. IMPLEMENTATION.....	125
7.6. ERWEITERUNGEN VON OQL.....	129
7.7. SCHNITTSTELLE ZU POLAR.....	130
<b>8. ZUSAMMENFASSUNG UND AUSBLICK.....</b>	<b>132</b>
<b>ANHANG A: BEISPIELE FÜR DIE UMSETZUNG.....</b>	<b>134</b>
<b>ANHANG B: SYNTAX FÜR JAVACC.....</b>	<b>136</b>
<b>ANHANG C: ABKÜRZUNGSVERZEICHNIS .....</b>	<b>139</b>
<b>ANHANG D: ABBILDUNGSVERZEICHNIS .....</b>	<b>141</b>
<b>ANHANG E: LITERATURVERZEICHNIS .....</b>	<b>142</b>

# 1. Einleitung

Moderne Softwaresysteme werden heutzutage überwiegend mit objektorientierten Programmiersprachen entwickelt. Zur Datenhaltung wird dabei meistens ein relationales Datenbanksystem verwendet. Das dadurch auftretende Problem ist, daß die innerhalb des gleichen Systems zu verwendenden Datenmodelle teilweise nicht verträglich sind, bedingt durch Einschränkungen, welche entweder bei der Programmiersprache oder bei dem relationalen Datenmodell vorhanden sind (Problem des sog. *impedance mismatch*). (teilweise adaptiert aus [IBL])

Deshalb wäre es eigentlich erforderlich, objektorientierte Datenbanksysteme anstelle von relationalen zu verwenden. Es gibt aber eine Menge Gründe, die Datenhaltung trotz Verwendung einer objektorientierten Programmiersprache mit relationalen Datenbanksystemen durchzuführen.

Einige dieser Gründe sind (entnommen aus [IBL97], Seite 7f):

- Der „Benutzer“ hat erst vor wenigen Jahren auf relationale Systeme umgestellt und daher hohe Investitionen getätigt, die er nicht einfach „wegwerfen“ will.
- „Daten sind teuer“. Die Übernahme von Daten von einem Datenbanksystem in ein anderes ist oft sehr teuer oder gar nicht möglich. Die Gründe hierfür sind unterschiedliche Datenschemata oder inkompatible Datenbankmodelle. Eine Nachbearbeitung „von Hand“ ist oft nicht bezahlbar oder aufgrund des Datenvolumens nicht durchführbar.
- Relationale Datenbanksysteme sind eigentlich auf allen Systemen bzw. Plattformen erhältlich, bei objektorientierten Datenbanken ist dies (noch) nicht der Fall. Durch die Standardisierung von relationalen Datenbanken (z.B. SQL) ist die Abhängigkeit von einem bestimmten System bzw. Hersteller weitaus geringer.
- Es besteht eine Unsicherheit bezüglich der Stabilität des Datenbankherstellers. Hier stellen sich beispielsweise Fragen wie:  
Gibt es den Hersteller in einigen Jahren überhaupt noch? Welche Planungen gibt es bezüglich der Weiterentwicklung des Datenbanksystems? etc.  
Bei den großen Herstellern relationaler Datenbanksysteme (z.B. IBM, ORACLE) ist diese Stabilität und Kontinuität größtenteils gegeben.
- Die Notwendigkeit, daß sowohl bereits existierende, wie auch neu zu entwickelnde Applikationen auf denselben Datenbestand zugreifen, erfordert meist eine Beibehaltung des bestehenden relationalen Datenmodells, insbesondere wenn gleichzeitig aus Programmen, die mit unterschiedlichen Programmiersprachen implementiert wurden, auf eine Datenbank zugegriffen werden soll, bieten die Hersteller von objektorientierten Datenbanken nur sehr schwache Unterstützung. So sind beispielsweise Schnittstellen für herkömmliche 3GL-Sprachen wie C, FORTRAN oder COBOL üblicherweise nicht verfügbar.

Es ergibt sich dadurch die Situation, die (von [Neu95]) wie folgt beschrieben wird:

*„Der durchgängigen Anwendungsentwicklung mit OOA, OOD und anschließender Implementation mit einer objektorientierten Programmiersprache steht die Notwendigkeit der Verwendung eines relationalen Datenbanksystems gegenüber“.*

Die Verbindung beider Konzepte ist jedoch nicht unproblematisch:

Das Zusammenspiel eines relationalen Datenbanksystems mit einer objektorientierten Programmiersprache kann dieser viel von ihren Vorteilen nehmen, schlimmstenfalls führt die Integration von relationalen Datenbankanweisungen in ein objektorientiertes Programm dazu, daß mit der objektorientierten Programmiersprache prozedurale Programme entwickelt werden, was zur Folge hat, daß das objektorientierte Konzept verwaschen wird.

Problembehaftet ist auch die Vermischung von Anwendungs- und Systemschicht, was bedeutet, daß der Softwareentwickler Teile der Systemschicht für die Bereitstellung der Daten durch das RDBS in das eigentliche Anwendungsprogramm mit aufnimmt. Anpassungen und Veränderungen der Systemschicht betreffen dann auch immer die Anwendungsprogramme.

Zur Vermeidung dieser Probleme, wurden sogenannte objektorientierte Datenbankmodelle bzw. *Persistenzframeworks* für die Koppelung zwischen dem objektorientierten- und dem relationalen Datenmodell entwickelt. Diese Persistenzframeworks werden als zusätzliche Schicht zwischen die relationale Datenhaltung und die objektorientierte Anwendungsentwicklung geschoben. Sie erweitern die objektorientierte Programmiersprache um ein objektorientiertes Datenbankmodell, das sowohl das zur Speicherung der Objekte verwendete relationale Datenmodell, wie auch die zugehörige Datenbanksprache (hier: SQL) vor dem Benutzer verbirgt. Dies ermöglicht die Abschirmung der Anwendungsentwicklung von den Zugriffen auf die relationale Datenbank, so daß sich der Benutzer ausschließlich im Datenmodell seiner objektorientierten Programmiersprache bewegen kann. Aus Sicht eines Anwendungsprogramms stellt das Framework also ein objektorientiertes Datenbanksystem zur Verfügung, das die Operationen der objektorientierten Welt automatisch in Operationen des darunterliegenden relationalen Datenbanksystems umsetzt und umgekehrt.

Ein solches objektorientiertes Datenbankmodell hat unter anderem die Aufgabe, den Benutzer bei der Suche nach persistent gespeicherten Objekten zu unterstützen. Dabei sollte die Suchbedingung in dem für ihn sichtbaren Datenmodell formulierbar sein, so daß der Benutzer nichts über die Abbildung des objektorientierten- auf das relationale Datenbankmodell wissen muß.

Es wäre konzeptuell ein Fehler, wenn der Benutzer SQL-Anweisungen für die Suche nach Objekten verwenden müßte, da er sich dadurch gezwungenermaßen mit der relationalen Abbildung der Objekte beschäftigen müßte.

Die heute am Markt verfügbaren Persistenzframeworks bieten die sogenannten assoziativen Objektanfragen an. Bei diesen Anfragen dienen Musterobjekte als Träger der Suchbedingung. Ausgehend von einem oder mehreren solchen Objekten wird automatisch eine Anfrage in SQL formuliert, die alle Objekte aus der Datenbank ermittelt, die so ähnlich sind wie die Musterobjekte. Assoziative Objektanfragen sind allerdings nicht übermäßig mächtig, bestimmte Anfragen können mit ihnen nicht erzeugt werden (bspw. Anfragen wie „Liefere alle Personen die keine Adresse haben“ sind nicht möglich, da die Attribute ignoriert werden, für die keine Angaben im Musterobjekt gemacht wurden. Eine Negation kann daher nicht ausgedrückt werden).

Ein Quasi-Standard für objektorientierte Datenbanksysteme ist von der Object Database Management Group unter dem Namen ODMG 2.0 verabschiedet worden. Er enthält u.a. die Definition einer objektorientierten Anfragesprache, welche sich zunehmend zum Standard für objektorientierte Anfragesprachen kristallisiert. Diese Sprache ist unter dem Namen OQL (Object Query Language) bekannt. Sie ist eine Erweiterung der bekannten Datenbanksprache SQL für relationale Systeme, um objektorientierte Konzepte. Der Vorteil dieser Anfragesprache ist, daß sie sich auf dem objektorientierten Datenmodell bewegt und relativ starke Ausdrucksmöglichkeiten bietet. Um anspruchsvolle, objektorientierte Anfragen an die Datenbank absetzen zu können, wäre es also sinnvoll, die oben angesprochenen Persistenzframeworks um die objektorientierte Anfragesprache OQL zu erweitern.

Diesem Ziel folgend, beschäftigt sich diese Diplomarbeit mit der Umsetzung von OQL-Anfragen für ein Persistenzframework mit dem Namen POLAR<sup>®</sup>. Ziel dieser Arbeit ist, demnach die Umsetzung von OQL-Anfragen in SQL-Anfragen, da nur unter Verwendung der Sprache SQL ein Zugriff auf das RDBS möglich ist. Informationen über die Abbildung von Objekten auf relationale Strukturen werden von dem Persistenzframework zur Verfügung gestellt. Es soll ermittelt werden, inwiefern OQL-Anfragen mit Hilfe eines Compilers direkt in SQL-Anfragen übersetzt werden können. Das praktische Interesse ist hierbei klar: Wenn OQL-Anfragen direkt in SQL übersetzt werden können, indem OQL-Anfragen mit Hilfe eines in das Persistenzframework integrierten Compilers übersetzt werden, so können Persistenzframeworks objektorientierte Anfragen ohne relevanten Performance-Einbruch unterstützen, da ja das relationale DBS weiterhin die erzeugte Anfrage optimiert. Somit wären die Persistenzframeworks den objektorientierten Datenbanksystemen in puncto Anfragemöglichkeiten nicht mehr unterlegen.

Die Arbeit beginnt - in Kapitel 2 - mit einer Darstellung des ODMG 2.0 Standards, der u.a. die Sprache OQL definiert. Dabei werden in Abschnitt 2.3.3. die hier aufgefundenen Mängel in der Sprachdefinition von OQL kritisch aufgezeigt. Im Anschluß daran wird angegeben, auf welche Teilmenge von SQL sich diese Arbeit bezieht, d.h. welche Teile von SQL für diese Arbeit verwendet werden können. Dies ist notwendig, da das - in Kapitel 4 - beschriebene Persistenzframework POLAR<sup>®</sup> auf beliebigen RDBS betrieben werden kann, so z.B. unter MS-Access, das nicht einmal eine vollständige SQL89-Unterstützung bietet.

In dem darauffolgenden Kapitel werden grundlegende Theorien des Compilerbaus - für die Entwicklung des Compilers der OQL-Anfragen in adäquate SQL-Anfragen übersetzt -

vorgestellt. Außerdem wird das Werkzeug Java Compiler Compiler beschrieben. Mit Hilfe dieses Werkzeugs wird - in Kapitel 6 - ein Parser generiert, der eine OQL-Anfrage in Form einer Zeichenkette in einen Syntaxbaum umwandelt.

Im Rahmen der darauf folgenden semantischen Analyse wird festgestellt, ob sich die OQL-Anfrage in eine einzige SQL-Anfrage übersetzen läßt. Ist dies der Fall, wird die Übersetzung durchgeführt, andernfalls muß die OQL-Anfrage durch einen eigenen OQL-Interpreter ausgewertet werden, indem kleinere SQL-Anfragen erzeugt und deren Ergebnisse dann verarbeitet werden.

Die Arbeit wird mit einer kurzen Zusammenfassung und einem kleinen Ausblick abgeschlossen.

Für die Implementierung dieser Arbeit wurde die objektorientierte Programmiersprache Java verwendet. Ein Grund dafür ist die stetig zunehmende Bedeutung dieser Sprache. Ein weiterer, wohl wichtigerer Grund, der für die Programmiersprache Java spricht, ist, daß diese Arbeit auf dem in Java entwickelten Persistenzframework POLAR<sup>®</sup> aufbaut und dieses um die objektorientierte Anfragesprache OQL erweitern soll. Durch die Verwendung einer anderen Programmiersprache als Java gäbe es Schnittstellen-Probleme mit dem übrigen POLAR<sup>®</sup>-Framework.

## 2. Der Standard ODMG 2.0

Objektorientierte Datenbanken sind nach objektorientierten Prinzipien aufgebaut, d.h. die Elemente einer objektorientierten Datenbank sind Objekte, deren Struktur durch Klassen definiert ist.

Damit nicht jeder Hersteller irgendwie objektorientierte Datenbanksysteme entwickelt, die mit denen anderer Hersteller nicht kompatibel sind, muß ein Standard entwickelt werden. Mit diesem Ziel, einen Standard für objektorientierte Datenbanken zu setzen und dessen Verbreitung zu fördern wurde die ODMG (=Object Database Management Group) als Vereinigung führender Datenbankhersteller gegründet. Als selbsternanntes Normungsgremium entwickelte sie ihren Standard für objektorientierte Datenbanken, der unter dem Namen ODMG bekannt ist und der aktuell in der Version 2.0 vorliegt.

Er besteht aus folgenden Komponenten:

1. *Object Model:*

Im Objektmodell werden die Bestandteile einer objektorientierten Datenbank definiert. Für die Erstellung einer solchen Datenbank wird ein Metamodell des internen Aufbaus festgelegt.

2. *Object Specification Languages:*

Der Standard definiert zwei Spezifikationsprachen, die *Object Definition Language (ODL)*, die zur Definition eines Datenbankschemas in Form von Klassen (Attributen und Methoden) dient, sowie die *Object Interchange Format Language (OIF)*, die dazu dient, den Datenbankzustand, also Daten und Schema-Information, in eine oder mehrere Dateien zu schreiben und von dort wieder zu lesen. Die OIF Language kann also unter anderem zum Austausch von Objekten zwischen mehreren Datenbanken verwendet werden.

3. *Object Query Language (OQL):*

OQL ist eine einfache, deklarative Sprache zum Zugriff auf Datenbanken, die auf dem objektorientierten Datenbankmodell basieren.

4. *Language Bindings:*

Zusätzlich zu den obigen drei Komponenten gibt es Definitionen für die Anbindung verschiedener objektorientierter Programmiersprachen an das ODMG Object Model. Bisher gibt es Sprachanbindungen für SmallTalk, Java und C++.

5. Querbezüge zu OMG und CORBA (siehe [Catt98], Seite 245ff).

Grundlage für die Entwicklung des ODMG 2.0 Standards war, laut [Catt98], der ANSI SQL-92 Standard [ISO], das OMG Object Model und die Interface Definition Language (IDL).

Das in dem folgenden Abschnitt beschriebene Object Model ist für das Verständnis der Sprache OQL notwendig, da es die Datentypen festlegt, die von OQL verwendet werden können. Im Anschluß an das Object Model werden - zur Vollständigkeit - die Object Specification Languages kurz angesprochen. Nachfolgend wird eine detaillierte Einführung in OQL gegeben.

### 2.1. Object Model

Das Object Model der ODMG definiert die Basiselemente einer objektorientierten Datenbank, indem es auf dem Object Model der OMG aufbaut und dieses um einige Elemente erweitert:

- Es gibt *Objekte* und *Literale*: Objekte haben eine über das gesamte Datenbanksystem eindeutige Objektidentität (OID). Literale werden durch ihren Wert (Inhalt) unterschieden.
- *Atomare Objekte* besitzen eine eindeutige OID, sie dürfen selbst keine Komponenten mit einer OID enthalten.
- Objekte können durch Zeichenketten benannt werden, allerdings muß ihr Name in der Datenbank eindeutig sein.
- Objekte können entweder *transient* (gehen beim Beenden der Anwendung verloren) oder *persistent* (existieren über die Laufzeit eines Programmes hinaus) gespeichert werden. Einstiegs-Punkte in eine objektorientierte Datenbank bilden sogenannte Wurzelobjekte („*root objects*“), von welchen man die restlichen Objekte der Datenbank erreichen kann. Den Wurzelobjekten werden Namen zugeordnet, über die sie angesprochen werden können.
- „*Zu den Eigenschaften der Objekte, die ihren Status definieren, gehören zusätzlich zu den Attributen auch Beziehungen zu anderen Objekten. Zwar gibt es auch im Objektmodell der OMG die Möglichkeit, Beziehungen zwischen Objekten durch Aggregations- oder Assoziationsmechanismen aufzubauen. Der Unterschied ist aber, daß eine solche Beziehung vom Datenbanksystem verwaltet wird und die Beziehung immer auch invers (d.h. in beiden Richtungen) vorhanden ist*“ (zitiert aus [Heu92]).
- Es gibt Kollektionen variabler Größen, die sowohl Objekte als auch Literale aufnehmen können. Alle Elemente einer Kollektion müssen den gleichen Typ aufweisen. Als Kollektionen gibt es Mengen (*Sets*), Multimengen (*Bags*), Listen (*Lists*), Felder (*Arrays*) und assoziative Felder (*Dictionary*s).
- Es gibt die durch den Tupel-Konstruktor (Verbund) zusammengesetzten Typen *Time*, *Date*, *Timestamp* und *Interval*.

- Literale sind die einfachen Datentypen (*Integer*, *Float*, *String*, ...), die zusammengesetzten Datentypen (*Date*, *Time*, ...) und alle durch Kollektionen (*Set*, *Bag*, ...) über Literalen und Strukturen (*structs*) erzeugten Datentypen die wiederum aus Literalen bestehen.
- Metadaten werden in einem *ODL Schema Repository* gespeichert. (Vgl. als Gegenstück das *IDL Interface Repository* in CORBA-Umgebungen)

## 2.2. Object Specification Languages

Wie bereits angesprochen unterstützt der ODMG-Standard zwei Spezifikations-sprachen, die Object Definition Language und die Object Interchange Format Language.

### 2.2.1. Object Definition Language (ODL)

Die Object Definition Language (ODL) ist eine Spezifikations-sprache, die zur Definition von Klassen in objektorientierten Datenbanken dient. Laut ODMG ist sie unabhängig von einer bestimmten Programmiersprachen, lehnt sich aber syntaktisch stark an C++ an. Sie unterstützt sämtliche Konzepte des Objekt-Modells, ist dabei aber nicht Turing-mächtig. Mit der ODL können die Attribute und Methoden von Klassen beschrieben werden. Die eigentliche Implementierung der Methoden bleibt jedoch einer herkömmlichen Programmiersprache überlassen.

*„Ziel der Entwicklung der Sprache ODL war, die Portabilität von Datenbankschemata unter ODMG konformen Datenbanken gewährleisten zu können“* (aus [Catt98], S.57).

Beispiel einer ODL Klassendefinition:

```
class Adresse
(
    extent Adressen)
{
    // Attribute
    attribute string ort;
    attribute string strasse;
    attribute long postleitzahl;

    // Methoden
    long getPostleitzahl();
    string getOrt();
}
```

```
class Person
(
    extent Personen)
{
    // Attribute
    attribute string name;
    attribute string vorname;
    relationship Adresse hatAdresse
        inverse Adresse::hatMieter

    // Methoden
    string getVorname();
    string getName();
}
```

Eine vollständige Beschreibung der Grammatik von ODL befindet sich in ([Catt98] S. 67ff).

### 2.2.2. Object Interchange Format Language (OIF)

Die Object Interchange Format Language (OIF) ist eine Sprache, mit der man den aktuellen Datenbankzustand, bestehend aus Daten und Schema-Informationen, in eine oder mehrere Dateien schreiben und von dort wieder auslesen kann. Die OIF Language wird hauptsächlich verwendet für den Austausch von Objekten zwischen Datenbanken und um den aktuellen Zustand des Datenbanksystems ausgeben zu können. Für eine detailliertere Beschreibung der OIF Language sei auf das Buch ([Catt98], S.72-.81) verwiesen.

### 2.3. OQL

OQL (siehe [Catt98], S.83-119) ist eine Sprache zum Zugriff auf die Elemente einer objektorientierten Datenbank. Sie kann sowohl zu Anfragen innerhalb eines Query-Interpreters als auch zum Datenbankzugriff innerhalb einer anderen Programmiersprache (wie etwa SmallTalk, Java oder C++) verwendet werden.

#### 2.3.1. Grundlagen

OQL ist keine berechnungs-vollständige Programmiersprache, wie etwa das geplante SQL3, sondern bietet lediglich eine Schnittstelle zum Zugriff auf eine objektorientierte Datenbank. Die notwendige Funktionalität zur Bearbeitung von Daten muß von der benutzen Programmiersprache selbst geleistet werden. Innerhalb eines OQL-Ausdrucks ist es jedoch möglich, Methoden von fachlichen Klassen aufzurufen, welche die notwendige Funktionalität realisieren können. Trotzdem ist OQL eine reine Anfragesprache, d.h. Veränderungen innerhalb einer Datenbank müssen über die Methoden der entsprechenden fachlichen Klassen erfolgen. Es gibt, im Gegensatz zu SQL, keine Update- oder Insert-Befehle.

OQL ist eine orthogonale Sprache. Die Operatoren der Sprache können - unter Beachtung des Typsystems - beliebig zusammengesetzt bzw. kombiniert werden.

Beispiele zweier einfacher OQL Anfragen:

Die folgende Anfrage liefert eine Liste von allen Personen, die in Entenhausen' wohnen, wobei die Liste nach den Namen der Personen aufsteigend sortiert ist.

```
select    *
from      Person p
where     p.adresse.ort like "Entenhausen"
order by  p.name
```

Die folgende Anfrage liefert den ersten Namen aus der Liste bestehend aus allen Personen die in 'Entenhausen' wohnen, wobei die Liste nach den Namen der Personen aufsteigend sortiert ist.

```
first(select      *
      from        Person p
      where       p.adresse.ort like "Entenhausen"
      order by   p.name).name
```

### 2.3.1.1. Objekte und Literale

Von OQL werden sowohl *Objekte* wie auch *Literale* unterstützt. Objekte haben eine eindeutige Objektidentität (OID) während Literale durch ihre Werte unterschieden werden und keine Objektidentität haben. Einfache (atomare) Literale sind Boolean, Integer, Float, Character und String. Sie lassen sich durch Verwendung der Kollektionen Set (Menge), Bag (Multimenge), List (Liste), Array (Feld) und durch den Verbund (Struct) zu komplexeren Literalen zusammenfassen.

Ist *a* ein einfaches (atomares) Literal, so ist *a* ein OQL-Ausdruck, dessen Wert das Literal selbst darstellt. Ist *a* ein Name eines Objekts, so ist *a* ein Ausdruck, der den mit der Entität verbundenen Wert zurückliefert.

Als einfache (atomare) Literale kennt OQL:

Objekt Literale:		nil
Bool'sche Literale:	Boolean	true, false
Ganzzahlige Literale:	Integer	z.B. 10, -5
Fließkomma Literale:	Float	z.B. 3.14, 314.16e-2
Zeichen Literale:	Character	z.B. 'a'
Zeichenketten Literale:	String	z.B. "string", 'string'

(Zeichenketten müssen (nach [Catt98], S.115) nicht unbedingt in doppelten, sondern können, wie in SQL auch, in einfachen Hochkommata stehen).

### 2.3.1.2. Grundlegendes über Anfragen

Als wichtiger Unterschied zwischen OQL und SQL ist festzuhalten, daß SQL als Anfrage-Ergebnis immer eine Menge von Tupeln einer Relation liefert, hingegen ist bei OQL die Art des Ergebnisses von der Anfrage selbst abhängig. Das Ergebnis einer OQL-Anfrage kann beispielsweise eine Kollektion, ein Literal, ein Verbund oder ein Objekt sein. Im Gegensatz zu SQL kann das Ergebnis einer OQL-Anfrage auch in erweiterter NF<sup>2</sup>-Form sein.

Ein weiterer Unterschied zu SQL ist, daß die Verwendung des Select From Where Konstrukts nicht zwingend vorgeschrieben ist. Die Anfrage 1+1 ist beispielsweise in OQL gültig, in SQL hingegen ungültig.

Beispiele für Anfrage-Ergebnisse:

1. *Kollektion über Objekten mit Identität:*

Eine Anfrage kann eine Kollektion bestehend aus Objekten (mit Identität) zurückliefern. Die folgende Anfrage liefert eine Kollektion bestehend aus Personen (Objekten) die den Namen Duck haben.

```
select * from Person p where p.name like "Duck"
```

2. *Kollektion über einem Verbund:*

Eine Anfrage kann eine Kollektion über einen Verbund zurück liefern. Das Ergebnis ist somit ein Literal. Die folgende Anfrage liefert eine Kollektion über einen Verbund, der aus zwei Komponenten *n* und *v* besteht, wobei *n* den Namen und *v* den Vornamen einer Person angibt.

```
select struct(n: p.name, v: p.vorname)
from   Person p
```

3. *Einzelnes Objekt mit Identität:*

Eine Anfrage kann ein einzelnes Objekt mit Identität zurückliefern. Man beachte, daß dies in SQL nicht möglich ist. Dort wird immer eine Menge als Anfrageergebnis zurückgeliefert, egal ob diese Menge nur aus einem einzigen Element besteht! Die Anfrage unten liefert die letzte Person (Objekt) aus der Liste der Personen, die nach den Namen sortiert sind. Das Ergebnis ist also keine Kollektion sondern ein einzelnes Objekt!

```
last(select  p
      from    Person p
      order by p.name)
```

4. *Einzelner Verbund:*

Eine Anfrage kann auch nur ein Literal zurückliefern.

Die nächste Anfrage gibt den ersten Namen aus der Liste der Personen zurück, die nach den Namen sortiert ist.

```
first(select  struct(n: p.name)
      from    Person p
      order by p.name)
```

5. *Ergebnis einer Anfrage muß nicht in erster Normalform sein:*

SQL liefert das Ergebnis einer Anfrage stets in erster Normalform. Das objektorientierte Datenbankmodell unterstützt das (erweiterte) NF<sup>2</sup>-Modell (Non First Normal Form Model). Dementsprechend ist es möglich, daß das Ergebnis einer OQL-Anfrage sich nicht in erster Normalform befindet, d.h. es sind ineinander geschachtelte Ergebnismengen möglich!

Die folgende Anfrage liefert alle gespeicherten Orte. Zu jedem Ort wird eine Multimenge aus Personen zurückgeliefert, die in dem jeweiligen Ort ihren 1. Wohnsitz haben.

```
select a.ort, (select *
              from   Person p
              where  p.wohnsitz=a.ort) as personen
from   Adresse a
```

Dementsprechend ist das Ergebnis vom Typ:

```
bag<struct<ort:string, personen:bag<p:Person>>>
```

Es liegt also eine geschachtelte Ergebnismenge vor, d.h. die Ergebnismenge ist nicht in erster Normalform.

### 2.3.1.2. Navigationsausdrücke

Ein wichtiges Merkmal des objektorientierten Datenbankmodells ist die Möglichkeit der Navigation zwischen Objekten. Darunter wird die Möglichkeit verstanden, solange von einem Objekt zu anderen Objekten entlang einfacher Beziehungen zu navigieren, bis man die Daten erreicht hat, die man braucht.

In OQL wird die Navigation zwischen Objekten durch die '.' bzw. die '->' Notation ausgedrückt.

Beispiele:

```
p.vater.mutter.adresse.ort
```

Mit Hilfe dieses Ausdrucks wird von der Person *p* zu dem Vater der Person *p* navigiert. Von diesem wird zu der Mutter des Vaters der Person *p* (Großmutter väterlicherseits der Person *p*) navigiert, und als Ergebnis der Anfrage wird von der Adresse der Großmutter (Objekt) das Attribut Ort zurückgeliefert.

```
select *
from   Person p
where  exists(p.freunde)
```

Obige Anfrage demonstriert eine Navigation bei einer 1:n-Beziehung: Eine Person hat mehrere Freunde, dementsprechend kann auf das Ergebnis der Navigation (*p.freunde*) nicht mehr mit dem '->' bzw. '.' Operator zugegriffen werden, da ja *p.freunde* kein einzelnes Objekt sondern eine Sammlung von Objekten (Kollektion) zurückliefert, d.h. der Ausdruck *p.freunde* kann nur dort verwendet werden, wo eine Kollektion als Typ erwartet wird.

Bei *n:m* Beziehungen kann die Navigation nicht verwendet werden. Hierfür gibt es - wie in SQL - die Select From Where - Anweisung, die weiter unten behandelt wird.

### 2.3.1.3. Das Typsystem

Die Sprache OQL verfügt nicht über ein eingebautes Typsystem, sondern benutzt das Typsystem, das durch die jeweilige Anbindungs-Umgebung definiert ist. Das heißt, in einer Java-Umgebung sind dies die Java-Datentypen, dementsprechend in einer SmallTalk-Umgebung die SmallTalk-Datentypen.

OQL ist eine getypte Sprache, das heißt, jeder Ausdruck hat einen Typ. Es gibt keine typenlosen Ausdrücke. Der Typ kann abgeleitet werden aus der Struktur des Ausdruckes mit Hilfe der Typendeklarationen des Schemas und aus den Typen von benannten Objekten und Literalen. Alle Ausdrücke können so bereits zur Compilezeit auf Typenkorrektheit überprüft werden.

Die Kompatibilität von Typen wird wie folgt rekursiv definiert (aus [Catt98], S. 112f):

1. Typ  $t$  ist kompatibel mit Typ  $t$
2. Falls Typ  $t$  kompatibel mit Typ  $t'$  ist, dann gilt:
  - $set(t)$  ist kompatibel mit  $set(t')$
  - $bag(t)$  ist kompatibel mit  $bag(t')$
  - $list(t)$  ist kompatibel mit  $list(t')$
  - $array(t)$  ist kompatibel mit  $array(t')$
3. Falls es einen Typ  $t$  gibt, so daß  $t$  ein Supertyp von  $t_1$  und  $t_2$  ist, dann sind  $t_1$  und  $t_2$  kompatibel.

Dies hat folgende Auswirkungen für die Kompatibilität:

- Literale und Objekte sind nicht miteinander kompatibel
- Atomare Literal-Typen sind nur dann miteinander kompatibel, wenn sie gleich sind. Insbesondere sind Fließkommazahlen (Float) und ganze Zahlen (Integer) *nicht* miteinander kompatibel!
- Objekte sind miteinander nur dann kompatibel, wenn sie einen gemeinsamen Supertyp haben.

Falls  $t_1, t_2, \dots, t_n$  kompatible Typen sind, dann existiert genau ein Typ  $t$ , für den gilt:

1.  $\forall i$  mit  $1 \leq i \leq n$  gilt:  $t=t_i$  oder  $t$  ist Supertyp von  $t_i$
2. Es gibt kein Typ  $t' \neq t$  der 1. erfüllt und Supertyp von  $t$  ist.

Dieser Typ  $t$  wird als  $\text{lub}(t_1, t_2, \dots, t_n)$  bezeichnet.

### 2.3.1.4. Sichtbarkeitsregeln

Variablen können nur im From-Teil oder im Group-Teil einer Select From Where Anweisung definiert werden. In beiden Fällen muß zwischen einer impliziten und einer expliziten Variablendeklaration unterschieden werden. Bei der impliziten Deklaration wird der Name der Datenquelle selbst als Iteratorvariable (Aliasvariable) verwendet. Bei der expliziten Deklaration wird ein Bezeichner angegeben, der als Iteratorvariable verwendet wird. Die Variablen werden verwendet, um per Pfadausdruck zwischen Objekten zu navigieren, oder um auf die Attribute und Methoden einer Klasse zugreifen zu können. Wenn keine Mehrdeutigkeit existiert, kann der Attributname bzw. der Methodenname direkt als Kurzschreibweise verwendet werden, ohne daß der Variablenname als qualifizierter Bezeichner benutzt wird.

Beispiele:

Explizite Variablendeklaration:

```
select *
from   Person p
where  p.name like "Duck"
```

Implizite Variablendeklaration:

```
select *
from   Person
where  Person.name like "Duck"
```

Implizite Variablendeklaration mit Kurzschreibweise:

```
select *
from   Person
where  name like "Duck"
```

#### 2.3.1.4.1. Deklarationen im From-Teil

Der Gültigkeitsbereich deklarerter Variablen im From-Teil reicht über alle Teile der Select From Where Anweisung, also über den Select, From, Where, Group by, Having und Order by-Teil. Auch in allen eingeschachtelten Teilen der Select From Where Anweisung ist die deklarierte Variable gültig.

#### 2.3.1.4.2. Deklarationen im Group-Teil

Wird der Group by-Teil einer Select From Where Anweisung verwendet, so wird automatisch der Bezeichner `partition` deklariert (s.u.). Darüber hinaus können explizit Attributnamen deklariert werden, um die Gruppierung zu charakterisieren. Diese Attributnamen - wie auch der Bezeichner `partition` - sind in dem zur Select From Where Anweisung gehörenden Having- und Select-Teil gültig, ebenso in den dazu eingeschachtelten Ausdrücken.

Im ODMG-Standard wird nicht erwähnt, ob die Verschattung von Bezeichnern erlaubt ist, d.h. ob Sichtbarkeitsbereich und Gültigkeitsbereich durch die Blockstruktur auseinanderklaffen können.

### 2.3.1.5. Select From Where

Die zentrale OQL-Anweisung ist - wie in SQL - die Select From Where Anweisung. Mit ihr kann eine Kollektion aus einer Datenbank ausgewählt werden, dabei kann die Kollektion entweder aus Elementen vom Typ Objekt, Literal oder Kollektion bestehen. Das Ergebnis einer Select From Where Anweisung ist je nach der speziellen Ausprägung der Anweisung entweder eine Liste (*list*), eine Menge (*set*) oder eine Multimenge (*bag*), in jedem Fall aber eine Kollektion.

In dem From-Teil der Anweisung werden die Kollektionen angegeben, aus denen die Daten ausgewählt werden sollen. Aus ihnen wird das kartesische Produkt gebildet. Die optionale Where-Bedingung, die vom Typ `Boolean` sein muß, dient als Filter, der jedes Element des kartesischen Produktes aussiebt, das die Where-Bedingung nicht erfüllt (Selektion).

Mit Hilfe des Select-Teils können die noch nicht ausgesiebten Elemente auf bestimmte Werte projiziert werden, die dann als Verbunde in einer Kollektion zurückgeliefert werden. Wird im Select-Teil ein `*` angegeben, so werden alle nicht ausgesiebten Elemente zurückgeliefert. Das Schlüsselwort `distinct` im Select-Teil entfernt alle Duplikate aus der Ergebniskollektion.

Die Kollektionen im From-Teil einer Select From Where Anweisung können auf drei Arten umbenannt werden:

1. *Umbenennung ohne Schlüsselwort:*

Wird im From-Teil nach dem Namen der Kollektion ein Bezeichner angegeben, so steht der Bezeichner als Iterator-Variable (auch Aliasname genannt) für den aktuellen Eintrag aus der Kollektion.

Bsp.:

```
select *
from   Person p
where  p.name like "D*"
```

Der Bezeichner `p` steht als Iteratorvariable für die aktuelle Person.

2. *Umbenennung mit dem Schlüsselwort 'as':*

Wird im From-Teil nach dem Namen der Kollektion das Schlüsselwort `as`, gefolgt von einem Bezeichner, angegeben, so steht der Bezeichner als Iterator-Variable für den aktuellen Eintrag aus der Kollektion.

Bsp.:

```
select *
from   Person as p
where  p.name like "D*"
```

Der Bezeichner `p` steht als Iteratorvariable für die aktuelle Person.

### 3. Umbenennung mit dem Schlüsselwort 'in':

Wird im From-Teil ein Bezeichner, gefolgt von dem Schlüsselwort `in`, angegeben, so wird der Bezeichner als Iterator-Variable für den aktuellen Eintrag der nach dem Schlüsselwort `in` angegebenen Kollektion verwendet.

Bsp.:

```
select *
from   p in Person
where  p.name like "D*"
```

Der Bezeichner `p` steht als Iteratorvariable für die aktuelle Person.

Es gibt also syntaktisch drei Möglichkeiten eine Iterator-Variable zu definieren - die Semantik der drei Methoden ist gleich.

Das Ergebnis einer Select From Where Anweisung ist in jedem Fall eine Kollektion. Der Typ der Kollektion bestimmt sich wie folgt:

1. Wird in der Select From Where Anweisung der Order by - Teil verwendet, so ist die Ergebnis-Kollektion vom Typ `list` (Liste).
2. Wird in dem Select-Teil der Select From Where Anweisung das Schlüsselwort `distinct` verwendet und enthält die Anweisung keinen Order by - Teil, so ist die Ergebnis-Kollektion vom Typ `set` (Menge).
3. Wird im Select-Teil das Schlüsselwort `distinct` nicht verwendet und hat die Select From Where Anweisung keinen Order by - Teil, so ist die Ergebnis-Kollektion vom Typ `bag` (Multimenge).

Aufgrund der Orthogonalität von OQL ist es möglich, eine Select From Where Anweisung in jedem Teil einer anderen Select From Where Anweisung zu schachteln.

### 2.3.1.6. Group by und Having

Eine Select From Where Anweisung kann um einen Group by-Teil erweitert werden. Der Group by Operator teilt das kartesische Produkt des From-Teils in Gruppen, sogenannte Partitionen, auf. Die Gruppen können selbst durch Ausdrücke definiert werden und durch die Angabe von Bezeichnern benannt werden. Wird kein Name für sie angegeben, so wird ein interner (für den Benutzer unbekannter) Systemname verwendet.

Alle Elemente, bei denen die Gruppen (d.h. die angegebenen Ausdrücke im Group by-Teil) den gleichen Wert haben, gehören zu der gleichen Gruppierung. Der Group by-Teil liefert

diese Gruppen als Ergebnis in einer Menge über einem Verbund, der als Komponentennamen die Namen der angegebenen Gruppen-Bezeichner und als Komponenteneinträge die Werte der ausgewerteten Ausdrücke enthält. Zudem enthält jeder Verbund einen impliziten Komponentennamen `partition`, der alle Elemente des From-Teils enthält, die zu der jeweiligen Gruppierung gehören.

Beispiel:

```
select  *
from    Person p
group by Duck:    p.name = "Duck",
          Dagobert: p.vorname = "Dagobert"
```

Die Anfrage liefert als Ergebnis:

```
set<struct(Duck: Boolean, Dagobert: Boolean,
          partition: bag<p:Person>)>
```

Die Bezeichner `Duck` und `Dagobert` definieren die Gruppierung.

In diesem Fall gibt es vier mögliche Gruppen:

```
{(Duck = true, Dagobert = true),
 (Duck = true, Dagobert = false),
 (Duck = false, Dagobert = true),
 (Duck = false, Dagobert = false)}
```

Die implizite Variable `partition` enthält alle Personen, die zu der jeweiligen Gruppierung gehören. Beispielsweise enthält die Gruppierung (`Duck = true, Dagobert = false`) in der Variable `partition` alle Personen, die den Namen `Duck` und nicht den Vornamen `Dagobert` haben.

Wird eine `Select`-Anweisung mit einem `Group by`-Teil verwendet, so kann zusätzlich ein `Having`-Teil verwendet werden. Die optionale `Having`-Bedingung, die vom Typ `Boolean` sein muß, dient als Filter, der alle Gruppierungen aussiebt, die von dem `Group-by` Teil zurückgeliefert werden und nicht die `Having`-Bedingung erfüllen.

Es folgt ein Beispiel für eine Anfrage mit `Group by`, `Having` und eingeschachtelten `Select From Where` Anweisungen:

```
select  p.adresse.strasse as Strassen,
        (select * from partition p) as Personen
from    (select * from Person a
        where a.getAlter(>17) as p
where   p.name like "Duck"
group by Strasse: p.adresse.strasse
having  count(select p from partition p)>=10
order  p.adresse.strasse desc
```

Diese Anfrage liefert alle Straßen mit den Personen, die in der jeweiligen gleichen Straße wohnen, den Namen Duck haben und volljährig sind. Dabei werden nur diejenigen Straßen mit den Personen geliefert, in denen mindestens 10 Personen den Namen Duck haben und volljährig sind. Die Ergebnisliste ist absteigend nach den Straßen sortiert.

Das Ergebnis der Anfrage ist somit:

```
list(struct(Strassen: String, Personen: Bag<p:Person>))
```

### 2.3.1.7. Order by

Jede Form der Select From Where - Anweisung, egal ob sie Where, Group by oder Having verwendet wird, kann um einen Order-by Teil ergänzt werden.

Sind  $e_1, e_2, \dots, e_n$  Ausdrücke, auf denen eine Ordnung definiert ist, und steht  $s$  für eine beliebige Select From Where Anweisung (evtl. mit Group by und Having), so liefert der Ausdruck  $s$  order by  $e_1, e_2, \dots, e_n$  eine sortierte Liste, welche die von der Select From Where Anweisung ausgewählten Elemente enthält. Die Liste wird durch  $e_1$  als erstes Sortierkriterium, durch  $e_2$  als zweites Sortierkriterium ... und durch  $e_n$  als n. Sortierkriterium sortiert, d.h. als erstes Unterscheidungskriterium dient das erste Sortierkriterium. Sind Elemente nach dem ersten Sortierkriterium  $e_1$  gleich, so werden diese nach dem zweiten Sortierkriterium  $e_2$  miteinander verglichen. Sind wiederum Elemente nach dem ersten und dem zweiten Sortierkriterium gleich, so werden diese nach dem dritten Sortierkriterium  $e_3$  miteinander verglichen, usw. .

Nach jedem dieser Sortierkriterien  $e_1, e_2, \dots, e_n$  kann eines der Schlüsselwörter `asc` oder `desc` stehen. Das Schlüsselwort `asc` gibt an, daß für dieses Sortierkriterium aufsteigend sortiert werden soll, das Schlüsselwort `desc` gibt an, daß absteigend sortiert werden soll. Wird bei einem Sortierkriterium keines der Schlüsselwörter angegeben, so wird die Sortierreihenfolge des vorherigen Sortierkriteriums verwendet. Für das erste Sortierkriterium wird per definitionem aufsteigend sortiert, wenn keines der Schlüsselwörter `asc` oder `desc` angegeben wird.

In SQL kann nur nach Spalten von Tabellen sortiert werden. In OQL ist der Order-by-Teil mächtiger, denn es kann nach beliebigen Funktionen sortiert werden. Die einzige Voraussetzung ist, daß die Sortierkriterien  $e_1, e_2, \dots, e_n$  vom Typ `Comparable` sind.

Beispiel:

```
select    *
from      AG
order by  AG.aktien*AG.aktien-AG.einlage
```

Beispiel einer illegalen Eingabe:

```
select  *
from    Person p
order by (select * from Adresse order by ort)
```

Diese Anfrage ist nicht erlaubt, denn als Sortierkriterium wird eine Liste verwendet. Listen sind nicht vom Typ `Comparable`, d.h. Listen können nicht miteinander verglichen werden; eine Sortierung ist also nicht möglich.

### 2.3.1.8. Elementare Ausdrücke

#### 2.3.1.8.1. Unäre Operatoren

OQL unterstützt die folgenden unären Operatoren:

1. Unäre Operatoren für ganze Zahlen: `+`, `-`
2. Unäre Operatoren für Fließkommazahlen: `+`, `-`
3. Unärer Operator für die bool'sche Negation: `not`

Ist  $e$  ein Ausdruck, und  $op$  einer der Operatoren  $\{+, -\}$ , so liefert der Ausdruck  $op\ e$  einen Ausdruck vom Typ `Integer`, wenn  $e$  vom Typ `Integer` ist, und `Float`, wenn  $e$  vom Typ `Float` ist. Ist  $op = not$ , und  $e$  vom Typ `Boolean`, so liefert  $op\ e$  einen Ausdruck vom Typ `Boolean`.

#### 2.3.1.8.2. Binäre Operatoren

OQL unterstützt die folgenden binären Operatoren:

1. Binäre Operatoren für ganze Zahlen: `+`, `-`, `*`, `/`, `mod`
2. Binäre Operatoren für Fließkommazahlen: `+`, `-`, `*`, `/`
3. Binäre relationale Operatoren: `=`, `!=`, `<`, `<=`, `>`, `>=`
4. Bool'sche binäre Operatoren: `and`, `or`

Sind  $e_1$  und  $e_2$  Ausdrücke vom Typ `Integer` oder `Float`, und ist  $op$  einer der Operatoren  $\{+, -, *, /\}$ , so liefert der Ausdruck  $e_1\ op\ e_2$  einen Ausdruck vom Typ `Integer` wenn  $e_1$  und  $e_2$  beide vom Typ `Integer` sind, oder einen Ausdruck vom Typ `Float`, wenn mindestens einer der beiden Operanden ( $e_1, e_2$ ) vom Typ `Float` ist.

Sind  $e_1$  und  $e_2$  Ausdrücke vom Typ `Integer` so liefert der Ausdruck  $e_1\ mod\ e_2$  einen Ausdruck vom Typ `Integer`, dessen Wert  $e_1$  modulo  $e_2$  ist.

Sind  $e_1$  und  $e_2$  Ausdrücke vom Typ `Boolean` und ist  $op$  einer der Operatoren  $\{and, or\}$  so liefert der Ausdruck  $e_1\ op\ e_2$  einen Ausdruck vom Typ `Boolean`.

Sind  $e_1$  und  $e_2$  kompatible Typen, wobei `Integer` und `Float` hier als kompatibel betrachtet werden, und ist `op` einer der Operatoren `{=, !=}` so liefert der Ausdruck  $e_1 \text{ op } e_2$  einen Ausdruck vom Typ `Boolean`. Wie die Gleichheit (`=`) und Ungleichheit (`!=`) von Objekten definiert wird, wird von der ODMG (in [Catt98]) nicht angegeben.

Die Gleichheit von Literalen wird wie folgt definiert:

- Verbunde (*structs*) sind gleich, wenn sie die gleichen Komponenten haben (d.h. der Name und Typ der jeweiligen Komponente muß gleich sein), und wenn die Werte gleicher Komponentennamen übereinstimmen.
- Mengen (*sets*) sind gleich, wenn sie die gleichen Elemente enthalten.
- Multimengen (*bags*) sind gleich, wenn sie die gleichen Elemente enthalten und diese gleich oft vorkommen.
- Listen (*lists*) und Felder (*arrays*) sind gleich, wenn sie die gleichen Elemente in der gleichen Reihenfolge enthalten.

Sind  $e_1$  und  $e_2$  kompatible, atomare Typen, wobei `Integer` und `Float` als kompatibel betrachtet werden, und ist `op` einer der Operatoren `{<, <=, >, >=}` so liefert der Ausdruck  $e_1 \text{ op } e_2$  einen Ausdruck vom Typ `Boolean`. Da `<` nicht auf Verbunde definiert ist, können in OQL keine der Typen `Date`, `Time`,... direkt miteinander verglichen werden.

Beispiele:

```
struct(a:1,b:2)=struct(b:2,a:1)
```

liefert das Literal `true` als Resultat.

```
struct(a:1,b:2)=struct(c:1,d:2)
```

liefert einen Syntaxfehler, da die Komponentennamen unterschiedlich sind.

```
struct(a:1,b:2)<struct(a:1,b:2)
```

liefert einen Syntaxfehler, da Verbunde nicht atomare Typen sind.

OQL läßt es zu, daß die Auswertung von bool'schen Ausdrücken sofort beendet werden kann, wenn das Resultat schon feststeht (sog. *short-cut-evaluation*).

OQL ist eine deklarative Sprache, deren Semantik es erlaubt, Ausdrücke für die Optimierung umzuordnen. Dies führt allerdings zu einer Art Nichtdeterminismus: Sollte ein Ausdruck bei vollständiger Auswertung einen Laufzeitfehler erzeugen, so kann es bei unvollständiger Auswertung von der Reihenfolge der Auswertung abhängen, ob der Fehler ausgelöst wird oder nicht. Da aber die Reihenfolge der Auswertung von der Optimierung abhängt, ist für den Anwender unbestimmt, ob der Ausdruck den Laufzeitfehler auslösen wird oder nicht.

### 2.3.1.8.3. Operatoren auf Zeichenketten

OQL stellt auch einige Operatoren zur Bearbeitung von Zeichenketten zur Verfügung.

Die binären Operatoren `+` und `||` ermöglichen beide gleichermaßen die Konkatenation von Zeichenketten. Sind  $e_1$  und  $e_2$  Ausdrücke vom Typ `String`, so liefert der Ausdruck  $e_1 + e_2$  und der Ausdruck  $e_1 || e_2$  einen Ausdruck vom Typ `String`, dieser entspricht genau der Konkatenation der beiden Ausdrücke  $e_1$  und  $e_2$ .

Um zu prüfen, ob ein Zeichen in einer Zeichenkette enthalten ist, gibt es den binären Operator `in`. Er benötigt als linken Operand ein Zeichen vom Typ `Character` und als rechten Operand eine Zeichenkette vom Typ `String`. Ist das Zeichen in der Zeichenkette enthalten, so liefert der Operator das Literal `true`, ansonsten das Literal `false`, das Ergebnis ist also ein Ausdruck vom Typ `Boolean`.

Ist der Ausdruck  $e_1$  vom Typ `String` und der Ausdruck  $e_2$  vom Typ `Integer`, so liefert der Ausdruck  $e_1[e_2]$  einen Ausdruck vom Typ `Character`. Der Wert des Ergebnisses entspricht dem Zeichen, das sich in der Zeichenkette  $e_1$  an der Position  $e_2+1$  befindet, d.h.  $e_2$  gibt den Index des gewünschten Zeichens an, wobei das erste Zeichen den Index Null hat.

Ist der Ausdruck  $e_1$  vom Typ `String` und sind die Ausdrücke  $e_2$  und  $e_3$  beide vom Typ `Integer`, so liefert der Ausdruck  $e_1[e_2:e_3]$  einen Ausdruck vom Typ `String`, der von der Zeichenkette  $e_1$  eine Teil-Zeichenkette zurückliefert. Von der Zeichenkette  $e_1$  werden die Zeichen von Position  $e_2+1$  bis einschließlich der Position  $e_3+1$  als Zeichenkette zurückgeliefert, d.h. zu beachten ist, daß sowohl bei  $e_2$  als auch bei  $e_3$  der Index für das erste Zeichen mit der Zahl Null beginnt.

Sind  $e_1$  und  $e_2$  Ausdrücke vom Typ `String`, so liefert  $e_1 \text{ like } e_2$  einen Ausdruck vom Typ `Boolean`. Der Operator `like` liefert das Literal `true`, wenn  $e_1$  dem Muster  $e_2$  entspricht und `false` wenn  $e_1$  nicht dem Muster  $e_2$  entspricht. Das Muster  $e_2$  darf sogenannte Wildcard-Zeichen enthalten.

Als Wildcards sind definiert:

Wildcard-Zeichen	Bedeutung
* , %	diese beiden Wildcards stehen als Ersatz für beliebig viele Zeichen oder für gar kein Zeichen.
_ , ?	diese beiden Wildcards stehen als Ersatz für genau ein beliebiges Zeichen.
\	dieses Zeichen dient als ESCAPE-Zeichen, d.h. das folgende Zeichen wird nicht als Wildcard sondern als normales Zeichen behandelt.

Beispiele:

<code>"Dago"+"bert"</code>	liefert die Zeichenkette "Dagobert"
<code>"Da"    "gobert"</code>	liefert die Zeichenkette "Dagobert"
<code>'o' in "Dagobert"</code>	liefert das Bool'sche Literal "true"
<code>"Dagobert"[2]</code>	liefert das Zeichen 'g'
<code>"Dagobert"[4:7]</code>	liefert die Zeichenkette "bert"
<code>"Dagobert" like "D_g?b*t%"</code>	liefert das Bool'sche Literal "true"

### 2.3.1.9. Operationen auf Kollektionen

#### 2.3.1.9.1. Quantoren

OQL unterstützt sowohl den Allquantor  $\forall$  als auch den Existenzquantor  $\exists$ .

Der Allquantor wird syntaktisch durch `for all x in e1:e2`, der Existenzquantor durch `exists x in e1:e2` ausgedrückt. Dabei steht `x` für einen Variablennamen, `e1` für eine Kollektion und `e2` für eine Bedingung, d.h. für einen Ausdruck vom Typ `Boolean`. Der Variablenname `x` dient als Iteratorvariable, die das jeweils aktuelle Element der Kollektion angibt; sie wird zur Formulierung der Bedingung `e2` verwendet.

Der Existenzquantor liefert genau dann das Literal `true` als Ergebnis, wenn mindestens ein Element in der Kollektion `e1` den Ausdruck `e2` erfüllt; ist dies nicht der Fall, so wird das Literal `false` zurückgeliefert.

Der Allquantor liefert genau dann das Literal `true` als Ergebnis, wenn alle Elemente in der Kollektion `e1` den Ausdruck `e2` erfüllen. Erfüllt ein einziges Element nicht die Bedingung `e2`, so wird das Literal `false` zurückgeliefert.

Beispiele:

```
for all x in Person: x.name like "Duck"
liefert true, wenn alle Personen den Namen "Duck" haben, sonst false

exists x in Person: x.name like "Duck"
liefert true, wenn es mindestens eine Person gibt, die den Namen "Duck" hat.
```

Zusätzlich zu dem angesprochenen Existenz- und Allquantor gibt es noch die Funktionen `exists(e)` und `unique(e)`. Bei beiden Funktionen muß `e` eine Kollektion bezeichnen, das Resultat der beiden Funktionen ist ein Ausdruck vom Typ `Boolean`.

Die `exists` Funktion liefert das Literal `true`, wenn die Kollektion mindestens ein Element enthält, ansonsten liefert sie `false`.

Die `unique` Funktion liefert das Literal `true`, wenn die Kollektion genau ein Element enthält, in allen anderen Fällen liefert sie `false`.

### 2.3.1.9.2. Aggregatfunktionen

Aggregatfunktionen können nur auf Kollektionen angewandt werden. OQL kennt fünf verschiedene Aggregatfunktionen: `count`, `max`, `min`, `sum`, `avg`

Die Funktion `count` kann auf beliebige Kollektionen angewandt werden. Sie liefert als Ergebnis eine Zahl, die angibt, wie viele Elemente die Kollektion hat. Das Ergebnis dieser Funktion ist somit eine ganze Zahl, also ein Ausdruck vom Typ `Integer`.

Die Funktionen `min`, `max`, `sum` und `avg` können nur auf Kollektionen über Ganz- oder Fließkommazahlen angewandt werden. Das Ergebnis der `min`-Funktion ist die kleinste Zahl, das der `max`-Funktion ist die größte Zahl, die in der Kollektion gespeichert ist. Die `sum`-Funktion liefert die Summe aus allen Einträgen der Kollektion, die `avg`-Funktion deren Durchschnitt.

Als Resultat liefern die Funktionen `min`, `max`, `sum` und `avg` einen Ausdruck vom Typ `Integer`, wenn die Funktionen auf eine Kollektion über ganzen Zahlen angewandt wird. Werden die Funktionen auf eine Kollektion über Fließkommazahlen angewandt, so liefern sie einen Ausdruck vom Typ `Float`.

Beispiele:

<code>min(list(5,3,1,6,1,2))</code>	liefert die ganze Zahl 1
<code>max(list(5,3,1,6,1,2))</code>	liefert die ganze Zahl 6
<code>sum(list(5,3,1,6,1,2))</code>	liefert die ganze Zahl 18
<code>avg(list(5,3,1,6,1,2))</code>	liefert die ganze Zahl 3
<code>avg(list(6,3,1,6,1,2))</code>	liefert die ganze Zahl 3
<code>min(list(0.5,0.7,1.3,2))</code>	liefert die Fließkommazahl 0.5
<code>count(select * from Adresse)</code>	gibt die Anzahl der Adressen an

Im Gegensatz zu OQL verwendet SQL die Aggregatfunktionen nicht funktional. Obige Anfrage lautet deshalb in SQL:

```
SELECT COUNT(*) FROM Adresse
gibt die Anzahl der Adressen an (SQL)
```

Damit sich OQL syntaktisch nicht zu stark von SQL unterscheidet, haben die Begründer von OQL eine Regel eingeführt, die es erlauben soll, daß auch in OQL die Aggregatfunktionen in nicht funktionaler Schreibweise verwendet werden können.

Die Regel der ODMG lautet (nach [Catt98] S.115), für

`op = {min, max, sum, avg, count}` :

<code>select count(*) from ...</code>	is equivalent to
<code>count(select * from ...)</code>	
<code>select op(query) from...</code>	is equivalent to
<code>op(select query from...)</code>	
<code>select op(distinct query) from...</code>	is equivalent to
<code>op(distinct(select query from...))</code>	

Probleme, die durch die Einführung dieser Regel entstehen, und die von der ODMG entweder nicht entdeckt oder nicht aufgeführt wurden, werden in Abschnitt 2.3.3. besprochen.

### 2.3.1.9.3. Test auf Mitgliedschaft

OQL bietet mit Hilfe des  $e_1 \text{ in } e_2$  Operators eine Möglichkeit zu prüfen, ob ein Element in einer Kollektion vorhanden ist oder nicht. Bei dieser Anfrage muß  $e_2$  eine Kollektion bezeichnen, und  $e_1$  ein Objekt oder ein Literal, das den gleichen Typ oder Subtyp hat, wie die Elemente der Kollektion  $e_2$ . Das Ergebnis dieser Anfrage ist das Literal `true` wenn das Element  $e_1$  in der Kollektion  $e_2$  vorhanden ist, andernfalls ist das Resultat `false`.

Beispiele:

```
2 in list(1,2,3)
liefert das Literal true
```

```
5.0 in list(1.2,2.,3.7)
liefert das Literal false
```

```
"Duck" in (select name from Person)
liefert true, wenn es eine Person mit Namen "Duck" gibt, ansonsten false.
```

### 2.3.1.9.4. Konkatenation von Kollektionen

OQL bietet mit Hilfe des Ausdrucks  $e_1 + e_2$  eine Möglichkeit an, Kollektionen miteinander zu konkatenieren. Bezeichnet  $e_1$  eine Kollektion vom Typ  $\text{list}\langle t_1 \rangle$ ,  $e_2$  eine Kollektion vom Typ  $\text{list}\langle t_2 \rangle$  und sind die Typen  $t_1$  und  $t_2$  kompatibel, so liefert der Ausdruck  $e_1 + e_2$  eine Liste vom Typ  $\text{list}(\text{lub}(t_1, t_2))$ . Bezeichnet  $e_1$  eine Kollektion vom Typ  $\text{array}\langle t_1 \rangle$ ,  $e_2$  eine Kollektion vom Typ  $\text{array}\langle t_2 \rangle$  und sind die Typen  $t_1$  und  $t_2$  kompatibel, so liefert der Ausdruck  $e_1 + e_2$  ein Feld vom Typ  $\text{array}(\text{lub}(t_1, t_2))$ .

Beispiel:

```
list(1,2,3)+list(3,4)
liefert die Liste (1,2,3,3,4)
```

## 2.3.1.10. Zugriff auf die Elemente einer Kollektion

### 2.3.1.10.1. Zugriff auf das i. Element

OQL erlaubt den Zugriff auf ein bestimmtes Element einer Kollektion. Voraussetzung dafür ist, daß die Kollektion eine Ordnung definiert, d.h. der Zugriff auf bestimmte Elemente einer Kollektion ist nur bei Listen und Feldern (*arrays*) möglich, da nur sie eine Ordnung definieren. Bei Mengen (*sets*) und Multimengen (*bags*) ist ein Zugriff auf ein bestimmtes Element der

Kollektion nicht möglich, da Mengen und Multimengen keine Reihenfolge für ihre Elemente festlegen.

Bezeichnet also  $e_1$  eine Kollektion vom Typ `list<t>` oder vom Typ `array<t>` (Liste bzw. Feld über Elementen vom Typ  $t$ ) und ist  $e_2$  eine ganze Zahl, so liefert der OQL-Ausdruck `e1[e2]` ein Element vom Typ  $t$ ; es handelt sich dabei um das  $e_2+1$ . Element der Kollektion, weil das erste Element mit der Zahl Null indiziert wird.

Beispiele:

```
list('a','b','c','d')[1]
liefert das Zeichen 'b'
```

```
(select * from Person order by name)[5]
liefert die 6. Person bei aufsteigend. alphabetischer Sortierung nach dem Namen
```

### 2.3.1.10.2. Zugriff auf das erste und letzte Element

Für den Zugriff auf das erste und letzte Element einer Kollektion gibt es die Funktionen `first(e)` und `last(e)`. Bezeichnet  $e$  eine Kollektion vom Typ `list<t>` oder vom Typ `array<t>` (Liste bzw. Feld über Elementen vom Typ  $t$ ), so liefert die Funktion `first(e)` das erste Element der Kollektion und `last(e)` das letzte Element.

Beispiele:

```
first(list('a','b','c','d'))
liefert das Zeichen 'a'
```

```
last(list('a','b','c','d'))
liefert das Zeichen 'd'
```

### 2.3.1.10.3. Zugriff auf einen Teil einer Kollektion

Es gibt in OQL auch die Möglichkeit eine Unterkollektion aus einer Kollektion zu bestimmen: Bezeichnet  $e_1$  eine Kollektion vom Typ `list<t>` oder vom Typ `array<t>` (Liste bzw. Feld über Elemente vom Typ  $t$ ) und sind  $e_2$  und  $e_3$  Ausdrücke vom Typ `Integer`, so liefert der OQL-Ausdruck `e1[e2:e3]` eine Unterkollektion über Elemente des Typs  $t$ . Diese Unterkollektion enthält dabei diejenigen Elemente aus  $e_1$  für deren Index gilt:  $e_2 \leq \text{Index} \leq e_3$ . Zu bemerken ist, daß die Elemente der Indexgrenzen auch in der Unterkollektion enthalten sind.

Beispiel:

```
list('a','b','c','d')[1:3]
liefert die Kollektion list('b','c','d')
```

### 2.3.1.11. Operationen auf Mengen

#### 2.3.1.11.1. Vereinigung, Durchschnitt und Differenz

OQL unterstützt die Vereinigung von Mengen durch den `union` Operator, den Durchschnitt zweier Mengen durch den `intersection` Operator und die Differenz zweier Mengen durch den `except` Operator:

Bezeichnet  $e_1$  einen Ausdruck vom Typ `bag<t1>` oder vom Typ `set<t1>`, und  $e_2$  einen Ausdruck vom Typ `bag<t2>` oder vom Typ `set<t2>` und sind  $t_1$  und  $t_2$  kompatibel, so liefert der Ausdruck  $e_1 \text{ op } e_2$  mit  $\text{op} = \{\text{union}, \text{intersection}, \text{except}\}$  dann einen Ausdruck vom Typ `set(lub(t1, t2))` wenn beide Ausdrücke vom Typ `set` sind bzw. den Typ `bag(lub(t1, t2))` wenn mindestens einer der beiden Operanden vom Typ `bag` ist, weil wenn die Typen der beiden Operanden verschieden sind, d.h. der eine Operand ein `bag` der andere ein `set` ist, der `set` vor der Ausführung des Mengenoperators in einen `bag` umgewandelt wird.

Beispiele:

```
bag(1,2,3,4,5) union bag(1,2,3,4)
liefert den bag(1,2,3,4,5,1,2,3,4)
```

```
bag(1,2,3,4,5) intersect bag(1,2,3,4)
liefert den bag(1,2,3,4)
```

```
bag(1,2,3,4,5) except bag(1,2,3,4)
liefert den bag(5)
```

```
bag(1,2,3,4,5) union set(1,2,3,4)
liefert den bag(1,2,3,4,5,1,2,3,4)
```

```
set(1,2,3,4,5) union set(1,2,3,4)
liefert den set(1,2,3,4,5)
```

#### 2.3.1.11.2. Inklusion

OQL unterstützt Operatoren zur Prüfung von Teilmengen-Beziehungen: Bezeichnet  $e_1$  einen Ausdruck vom Typ `bag<t1>` oder vom Typ `set<t1>`, und  $e_2$  einen Ausdruck vom Typ `bag<t2>` oder vom Typ `set<t2>` und sind  $t_1$  und  $t_2$  kompatibel, so liefert der Ausdruck  $e_1 \text{ op } e_2$  für  $\text{op} = \{>, \geq, <, \leq\}$  ein Ausdruck vom Typ `Boolean`. Wenn die Typen der beiden Operanden verschieden sind, d.h. der eine Operand ein `bag` der andere ein `set` ist, so wird der `set` vor der Ausführung der Inklusion in einen `bag` umgewandelt.

Der Ausdruck  $e_1 \leq e_2$  liefert `true`, wenn  $e_1$  eine Teilmenge von  $e_2$  ist,  $e_1 < e_2$  liefert `true`, wenn  $e_1$  eine echte Teilmenge von  $e_2$  ist, wenn also  $e_1$  eine Teilmenge von  $e_2$  ist und  $e_1 \neq e_2$  ist.  $e_1 \geq e_2$  liefert `true`, wenn  $e_1$  eine Obermenge von  $e_2$  ist,  $e_1 > e_2$  liefert `true`, wenn  $e_1$  eine echte Obermenge von  $e_2$  ist.

Beispiele:

```
bag(1,2,3,4) < bag(4,1,5,6,2,3,4)
liefert das Literal true
```

```
bag(1,2,3,4) > bag(4,2,1,3)
liefert das Literal false
```

### 2.3.1.12. Konvertierungen

#### 2.3.1.12.1. Umwandlung einer Liste in eine Menge

OQL stellt eine Funktion mit Namen `listtoiset` für die Umwandlung einer Liste in eine Menge zur Verfügung. Ist `e` ein Ausdruck vom Typ `list` (Liste), so liefert die Funktion `listtoiset(e)` einen Ausdruck vom Typ `set` (Menge). Diese Funktion konvertiert die Liste in eine Menge (set), indem alle Elemente der Liste in die Menge aufgenommen werden. Durch die Aufnahme in die Menge werden auf Grund der Eigenschaften einer Menge automatisch Duplikate entfernt.

Beispiel:

```
listtoiset(list(1,2,3,2,1))
liefert die Menge set(1,2,3)
```

#### 2.3.1.12.2. Umwandlung einer Kollektion in ein Element

In OQL gibt es die Möglichkeit, durch die Funktion `element`, eine Kollektion in ein einzelnes Element umzuwandeln: Ist `e` ein Ausdruck, der eine Kollektion vom Typ `t` bezeichnet, so liefert die Funktion `element(e)` genau dann ein Element (Objekt oder Literal) vom Typ `t`, wenn die Kollektion aus nur einem einzigen Element besteht. Besteht die Kollektion aus mehreren Elementen bzw. aus keinem Element, so wird ein Laufzeitfehler ausgelöst.

Beispiel:

```
element(list(1))
liefert das Literal 1
```

#### 2.3.1.12.3. Entfernen von Duplikaten

OQL erlaubt es, durch die Funktion `distinct` Duplikate in einer Kollektion zu entfernen. Ist `e` ein Ausdruck vom Typ `set(t)` (Menge) oder `bag(t)` (Multimenge), so liefert die Funktion `distinct(e)` ein Ergebnis vom Typ `set(t)` (Menge). Das Ergebnis der Funktion ist gleich der übergebenen Kollektion ohne doppelt vorkommende Elemente.

Ist  $e$  ein Ausdruck vom Typ `list(t)` (Liste) oder `array(t)` (Feld), so liefert die Funktion `distinct(e)` ein Ergebnis vom Typ `list(t)` bzw. `array(t)`. Das Ergebnis der Funktion entspricht der übergebenen Kollektion ohne doppelt vorkommende Elemente. Mehrfach auftretende Elemente werden entfernt, indem nur das Element, das zuerst auftritt, in der Ergebnis-Kollektion übernommen wird.

Beispiele:

```
distinct(list(1,5,4,2,3,5,1,2,1))  
liefert die Liste list(1,5,4,2,3)
```

```
distinct(bag(1,2,3,4,3,2,1))  
liefert die Menge set(1,4,2,3)
```

```
distinct(array(1,2,3,3,2,1))  
liefert das Feld array(1,2,3)
```

### 2.3.1.12.4. Einebenen von Kollektionen aus Kollektionen

Besteht eine Kollektion aus Elementen, die wiederum aus Kollektionen über Elementen bestehen, so kann diese Struktur mit Hilfe der Funktion `flatten` verflacht werden:

Ist  $e$  ein Ausdruck vom Typ `collection<collection<t>>`, so liefert die Funktion `flatten(e)` ein Ergebnis vom Typ `collection<t>`, d.h. die Funktion `flatten` löst auf erster Ebene die Hierarchie bestehend aus Kollektionen über Kollektionen auf.

Genauer:

Ist  $e$  ein Ausdruck vom Typ `collection1(collection2<t>)`, so liefert `flatten(e)` folgendes Resultat:

- Falls `collection2` ein `set` (bzw. `bag`) ist, dann wird die Vereinigung aus allen Elementen von `collection2`, die in diesem Fall vom Typ `set<t>` (bzw. `bag<t>`) sind, gebildet. Das Ergebnis der `flatten` Funktion ist dann vom Typ `set<t>` (bzw. `bag<t>`).
- Falls `collection2` eine `list` (bzw. `array`) ist, und `collection1` eine `list` (bzw. `array`) ist, dann werden alle Elemente von `collection1` miteinander konkateniert. Das Ergebnis der `flatten` Funktion ist dann vom Typ `collection2<t>`.
- Falls `collection2` eine `list` (bzw. `array`) ist, und `collection1` ein `set` (bzw. `bag`) ist, dann werden die Listen oder Arrays in Mengen `sets` (bzw. Multimengen `bags`) umgewandelt. Danach werden alle diese Mengen `sets` (bzw. Multimengen `bags`) miteinander vereinigt. Das Ergebnis der `flatten` Funktion ist dann vom Typ `set<t>` (bzw. `bag<t>`).

Beispiele:

```
flatten(list(bag(1,2), bag(2,3), bag(2,3,4)))  
liefert die Multimenge bag(1,2,2,3,2,3,4)
```

```
flatten(list(set(1,2), set(2,3), set(2,3,4)))  
liefert die Menge set(1,2,3,4)
```

```
flatten(list(list(1,2), list(2,3), list(2,3,4)))  
liefert die Liste list(1,2,2,3,2,3,4)
```

```
flatten(list(array(1,2), array(2,3), array(2,3,4)))  
liefert das Feld array(1,2,2,3,2,3,4)
```

```
flatten(set(list(1,2), list(2,3), list(2,3,4)))  
liefert die Menge set(1,2,3,4)
```

### 2.3.1.13. Erzeugen von Objekten, Kollektionen und Verbunden

OQL kennt keine Update- oder Insert- Befehle wie sie SQL kennt. Es bietet aber die Möglichkeit an, Objekte und Kollektionen mittels Konstruktoren zu erzeugen. Leider wird in der Sprachdefinition [Catt98] nicht erwähnt, ob diese transient (d.h. beim Beenden der Anwendung wieder gelöscht werden) oder persistent (d.h. über die Laufzeit des Programms hinaus existent sind) gespeichert werden (Eine Diskussion über derartige Probleme von OQL findet sich in Abschnitt 2.3.3. wieder).

#### 2.3.1.13.1. Erzeugen von Objekten

OQL erlaubt des Erstellen von Objekten. Ist  $k$  ein Name einer Klasse, sind  $p_1, p_2, \dots, p_n$  Attribute (Namen) der Klasse mit den Typen  $t_1, t_2, \dots, t_n$  und sind  $e_1, e_2, \dots, e_n$  Ausdrücke mit den Typen  $t_1, t_2, \dots, t_n$ , so liefert der Ausdruck  $k(p_1:e_1, p_2:e_2, \dots, p_n:e_n)$  ein Ergebnis vom Typ  $k$ .

Beispiel:

```
Person(name: "Duck", vorname: "Dagobert")  
erzeugt ein Objekt der Klasse Person, bei dem das Attribut name mit "Duck",  
das Attribut vorname mit "Dagobert" initialisiert wird.
```

#### 2.3.1.13.2. Erzeugen von Kollektionen

Kollektionen können über Objekte, Literale und Kollektionen erzeugt werden. Zu beachten ist, daß Kollektionen nicht über beliebige Elemente erzeugt werden können, d.h. die Typen aller Elemente einer Kollektion müssen zueinander kompatibel sein.

Sind  $e_1, e_2, \dots, e_n$  kompatible Ausdrücke mit den Typen  $t_1, t_2, \dots, t_n$ , und ist  $op$  ein Element der Menge  $\{set, list, bag, set, array\}$ , so liefert der Ausdruck  $op(e_1, e_2, \dots, e_n)$  einen Ausdruck vom Typ  $t = op(lub(t_1, t_2, \dots, t_n))$ .

Sind  $e_1, e_2, \dots, e_n$  kompatible Ausdrücke mit den Typen  $t_1, t_2, \dots, t_n$ , und ist der Operator nicht genannt, so liefert der Ausdruck  $(e_1, e_2, \dots, e_n)$  einen Ausdruck vom Typ  $t = list(lub(t_1, t_2, \dots, t_n))$ .

Sind die zwei Ausdrücke  $e_1$  und  $e_2$  beide vom Typ `Integer` oder beide vom Typ `Character`, so liefert der Ausdruck `list(e1, e2)` und der Ausdruck  $(e_1, e_2)$  ein Ausdruck vom Typ  $t = list(Integer)$  oder vom Typ  $t = list(Character)$ .

Beispiele:

<code>set(1, 2, 3, 2)</code>	liefert die Menge <code>set(1,2,3)</code>
<code>bag(1, 2, 3, 2)</code>	liefert die Multimenge <code>bag(1,2,3,2)</code>
<code>(1, 2, 3, 2)</code>	liefert die Liste <code>list(1,2,3,2)</code>
<code>(1..5)</code>	liefert die Liste <code>list(1,2,3,4,5)</code>
<code>list('b'..'e')</code>	liefert die Liste <code>list('b','c','d','e')</code>

### 2.3.1.13.3. Erzeugen von Verbunden

Sind  $p_1, p_2, \dots, p_n$  Komponentennamen, und  $e_1, e_2, \dots, e_n$  Ausdrücke mit den Typen  $t_1, t_2, \dots, t_n$ , so erzeugt der Ausdruck `struct(p1:e1, p2:e2, ..., pn:en)` einen Ausdruck vom Typ `struct(p1:t1, p2:t2, ..., pn:tn)`.

Beispiel:

```
struct(name: "Duck", vorname: "Dagobert")
erzeugt einen Verbund mit den zwei Komponentennamen name und vorname. Der
Wert des Komponentennamens name wird mit "Duck", der Wert des
Komponentennamens vorname wird mit "Dagobert" initialisiert.
```

Die Projektion auf mehrere Werte in einem Select-Teil ist äquivalent zur Erzeugung eines Verbundes mit den entsprechenden Komponentenwerten:

Beispiel:

```
select ort as o, strasse as s from Adresse

ist gleichbedeutend mit:

select struct(o: ort, s: strasse) from Adresse
```

### 2.3.1.14. Methoden- und Funktionsaufrufe

Ein Grundprinzip der objektorientierten Programmierung ist, daß Attribute und Methoden einer Klasse fest miteinander verbunden sind, d.h. daß Daten und Funktionen immer zusammengehören und nicht getrennt voneinander betrachtet werden dürfen. Um diesem Grundprinzip treu zu bleiben, erlaubt OQL zusätzlich zu dem Zugriff auf Attribute auch das Ausführen von Methoden einer Klasse. So können in OQL Methoden von Klassen überall dort aufgerufen werden, wo der Ergebnistyp der Methode mit dem erwarteten Typ in der OQL-Anfrage übereinstimmt.

Prinzipiell wird zwischen Methodenaufrufen mit und ohne Parametern unterschieden:

1. Die Notation, um Methoden ohne Parameter aufzurufen, entspricht der Notation, um auf Attribute einer Klasse zuzugreifen (Pfad-Navigation). Optional kann nach dem Methodennamen noch eine öffnende Klammer, gefolgt von einer schließenden angebracht werden.
2. Methoden mit Parametern werden wie Methoden ohne Parameter aufgerufen, allerdings ist nach dem Methodennamen eine Liste von evtl. mehreren Parametern notwendig, die durch Kommata getrennt und von runden Klammern umgeben sind.

Methoden können sowohl einfache Objekte, Literale als auch Kollektionen zurückliefern. Liefert eine Methode keinen Wert zurück, so verwendet OQL als Ergebnis der Methode das `nil` Objekt. Methodenaufrufe können auch in komplizierte Navigationsausdrücke eingebettet werden.

Beispiele:

```
p.getAdresse().getPostleitzahl() = 88239
```

Angenommen `p` sei ein Objekt der Klasse `Person`, dann liefert die Methode `getAdresse()` die Adresse der Person. Der Aufruf der Methode `getPostleitzahl()` gibt dann die Postleitzahl der Adresse zurück.

```
p.getAdresse().getPostleitzahl = 88239
```

liefert das gleiche Ergebnis wie oben (Methodenaufrufe können syntaktisch wie Attributzugriffe verwendet werden).

Funktionsaufrufe werden hier nicht betrachtet, denn im folgenden wird OQL in Verbindung mit der Programmiersprache Java untersucht. Da in Java keine Funktionsaufrufe möglich sind (es gibt nur Methodenaufrufe), kann es logischerweise in OQL auch keine Funktionsaufrufe geben.

Durch Methodenaufrufe kommt nun eine gewisse Form der Polymorphie mit ins Spiel, denn nach den Prinzipien der objektorientierten Programmierung muß bei einem Methodenaufruf immer die richtige Ausprägung der Methode aufgerufen werden, d.h. es kann nicht zur

Compilezeit bereits feststehen, welche Methode aufgerufen werden muß, sondern erst zur Laufzeit. Deshalb können Methodenaufrufe erst zur Laufzeit gebunden werden (*late binding*):

Beispiel:

Angenommen die Klassen AG und GmbH sind Subklassen der Klasse Kapitalgesellschaft und sie besitzen alle eine eigene Methode `getFirma`, welche die jeweilige Firmierung des Handelsbetriebs liefert. Dann muß bei folgender Anfrage je nachdem, ob Objekt `g` eine Kapitalgesellschaft, eine GmbH oder eine AG ist, die Methode `getFirma` der Klasse Kapitalgesellschaft, GmbH oder AG aufgerufen werden.

```
select g.getFirma()  
from kapitalgesellschaften g
```

Um gegebenenfalls die Klasse explizit angeben zu können, unterstützt OQL auch die explizite Angabe des Klassennamens, per Cast-Operator. Dadurch können Anfragen, die das Typsystem zur Compilezeit nicht zulassen würde, gestellt werden. Liegt zur Laufzeit aber nicht die angegebene Klasse vor, so wird ein Laufzeitfehler generiert.

Beispiel:

```
select ((AG)g).getVorstand()  
from kapitalgesellschaften g
```

### 2.3.1.15. Die Behandlung von nicht definierten Objekten

Die Regeln, wie mit nicht definierten Objekten umgegangen wird, lauten:

1. Die Operatoren `.` und `->` liefern als Ergebnis "nicht definiert" (`UNDEFINED`) wenn sie auf einen "nicht definierten" linken Operanden angewandt werden, d.h. wird auf ein Attribut oder eine Methode eines unbestimmten Objektes (z.B. `nil` - Objekt oder `UNDEFINED`) zugegriffen, so ist das Resultat `UNDEFINED`.
2. Die Vergleichsoperatoren (`=`, `!=`, `<`, `>`, `<=`, `>=`) liefern `false` als Ergebnis, wenn entweder einer oder beide Operanden "nicht definiert" (`UNDEFINED`) sind.
3. Es gibt zwei eingebaute Funktionen: `is_undefined` und `is_defined`. Die Funktion `is_undefined(UNDEFINED)` liefert `true`, die Funktion `is_defined(UNDEFINED)` liefert `false`.
4. Jede andere Operation mit "nicht definierten" (`UNDEFINED`) Operanden führt zu einem Laufzeitfehler.

### 2.3.1.16. Benannte Anfragen

In OQL gibt es die Möglichkeit, Query-Anfragen persistent zu speichern, ähnlich wie dies mit den Views bei SQL möglich ist.

Ist `id` ein Bezeichner, `e` ein Ausdruck und sind  $x_1, x_2, \dots, x_n$  freie Variablen im Ausdruck `e`, so speichert der Ausdruck `define [query] id(x1, x2, ..., xn) as e(x1, x2, ..., xn)` die Anfragefunktion mit dem Namen `id` in die Datenbank. Dabei darf `id` nicht den Namen einer in der Datenbank gespeicherten Klasse, Methode oder Funktion haben.

Die benannten Query-Anfragen werden bei jeder Anfrage stets neu kompiliert und ausgewertet, wie dies bei dem View-Mechanismus in SQL üblich ist.

Sie werden persistent gespeichert, und bleiben solange aktiv, bis sie überschrieben oder durch den Befehl `'delete definition id'` gelöscht werden. Query-Definitionen dürfen nicht überladen werden. Wird dies jedoch versucht, so wird die bisherige Query-Definition durch die neue Query-Definition überschrieben.

Beispiele für eine benannte Anfragen:

```
define query Stadt(x) as
  select  a.ort
  from    Adresse a
  where   a.ort=x
```

```
define allChilds as
  select *
  from   Person p
  where  p.getAlter() < 18
```

### 2.3.2. Informelle Syntaxbeschreibung der ODMG

Die ODMG verwendet eine Art Backus-Naur-Form zur Beschreibung von Grammatiken:

- { Symbol }      Symbol ist eine Iteration bestehend aus 0 oder n Symbolen.
- [ Symbol ]      Symbol ist optional, kann entfallen. (nicht verwechseln mit [ ])
- **Schlüsselwort**    Terminalzeichen der Grammatik

Die ODMG gibt die Syntax der Sprache OQL, in einer Art informellen Backus-Naur-Form (aus [Catt98], S. 115-119), wie folgt an:

#### Axioms

```
query_program ::= {define_query;} query
define_query  ::= define identifier as query
```

#### Basic

```
query ::= nil
query ::= true
query ::= false
query ::= integer_literal
query ::= float_literal
query ::= character_literal
query ::= string_literal
query ::= entry_name
query ::= query_name
query ::= bind_argument
query ::= from_variable_name
query ::= (query)
```

#### Simple Expressions

```
query ::= query + query
query ::= query - query
query ::= query * query
query ::= query / query
query ::= - query
query ::= query mod query
query ::= abs(query)
query ::= query || query
```

### Comparison

```
query ::= query comparison_operator query
query ::= query like string_literal
comparison_operator ::= =
comparison_operator ::= !=
comparison_operator ::= >
comparison_operator ::= <
comparison_operator ::= >=
comparison_operator ::= <=
```

### Boolean Expression

```
query ::= not query
query ::= query and query
query ::= query or query
```

### Constructor

```
query ::= type_name([query])
query ::= type_name(identifier: query {,identifier:query})
query ::= struct(identifier: query {,identifier:query})
query ::= set([query {,query}])
query ::= bag([query {,query}])
query ::= list([query {,query}])
query ::= (query, query {,query})
query ::= [list](query..query)
query ::= array([query {,query}])
```

### Collection Expression

```
query ::= for all identifier in query:query
query ::= exists identifier in query:query
query ::= exists(query)
query ::= unique(query)
query ::= query in query
query ::= query comparison_operator quantifier query
quantifier ::= some
quantifier ::= any
quantifier ::= all
query ::= count(query)
query ::= count(*)
query ::= sum(query)
query ::= min(query)
query ::= max(query)
query ::= avg(query)
```

### Accessor

```
query ::= query dot attribute_name
query ::= query dot relationship_name
query ::= query dot operation_name(query {,query})
dot ::= . | ->
query ::= *query
query ::= query [query]
query ::= query [query:query]
query ::= first(query)
query ::= last(query)
query ::= function_name([query {,query}])
```

### Select Expression

```
query ::= select [distinct] projection_attributes
           from variable_declaration
           {,variable_declaration}
           [where query]
           [group by partition_attributes]
           [having query]
           [order by sort_criterion {,sort_criterion}]
projection_attributes ::= projection {,projection}
projection_attributes ::= *
projection ::= query
projection ::= identifier:query
projection ::= query as identifier
variable_declaration ::= query [as] identifier]
partition_attributes ::= projection {,projection}
sort_criterion ::= query [ordering]
ordering ::= asc
ordering ::= desc
```

### Set Expression

```
query ::= query intersect query
query ::= query union query
query ::= query except query
```

### Conversion

```
query ::= listtaset(query)
query ::= element(query)
query ::= distinct(query)
query ::= flatten(query)
query ::= (class_name)query
```

### Operator Priorities

```
( ) [ ] . ->
not -(unary) +(unary)
in
* / mod intersect
+ - union except ||
< > <= >= < some < any < all (etc. ...)
= != like
and exists for all
or
.. :
,
(identifizier) (cast-Operator)
order
having
group by
where
from
select
```

Die Position der Operatoren in obiger Liste gibt deren Rangfolge an. Der Operator `select` hat beispielsweise die geringste Priorität, die Klammer repräsentiert den Operator mit höchster Priorität.

Eine OQL-Grammatik, welche die Operator-Prioritäten berücksichtigt und keine Links-Rekursionen enthält, wird in Kapitel 6 dieser Arbeit präsentiert.

### 2.3.3. Syntaktische und semantische Lücken von OQL

Die informelle Syntaxbeschreibung der Sprache OQL (s.o.) wurde bewußt aus der Sprachdefinition der ODMG [Catt98] entnommen. Damit kann verdeutlicht werden, daß es auf der einen Seite an einer vollständigen sowie fehlerfreien Beschreibung der Syntax, sowie auf der anderen Seite in einigen Punkten an einer halbwegs eindeutigen Definition der Semantik fehlt.

Die folgende Liste soll einige dieser Mängel belegen:

- In der Liste der Operator-Prioritäten taucht ein unärer Operator `+` auf, welcher laut Syntaxdefinition gar nicht vorhanden ist.
- Über die Priorität des unären Operators `*`, der in der BNF-Syntaxdefinition<sup>1</sup> auftaucht, wird dagegen nichts ausgesagt. Auch über die Semantik dieses unären Operators wird geschwiegen.
- In [Catt98] S.98 ist die Rede von einem unären Operator `abs` welcher z.B. in folgender Form zu verwenden sei: `abs -1`. Schaut man sich hingegen die BNF-Syntaxdefinition an, so fällt auf, daß dort `abs` als Funktion definiert wird, also in der folgenden Form zu verwenden ist: `abs(-1)`. Für diese Sicht spricht auch, daß in der Liste der Operator-Prioritäten `abs` gar nicht auftaucht.
- Die ODMG<sup>2</sup> definiert eine Fließkommazahl als Mantisse und Exponent, wobei der Exponent optional ist. Dazu werden zwei Beispiele gegeben: `3.14` und `314.16e-2`. Offen gelassen wird aber, ob Ausdrücke wie beispielsweise `314.` oder `.1` zugelassen sind.
- In ([Catt98], S.99) heißt es:  
*"If  $s$  is an expression of type string, and  $i$  is an expression of type integer, then  $S_i$  is an expression of type character whose value is the  $i+1$ th character of the string."* Abgesehen davon, daß es wohl `si` statt `Si` heißen sollte, soll dies die Syntax beschreiben, mit der man einen String indizieren kann. Mit den in der Syntaxdefinition beschriebenen Regeln läßt sich dieser Ausdruck aber nicht produzieren. Entweder es fehlt dort eine Produktionsregel, oder aber es liegt ein Schreibfehler in oben genanntem Abschnitt vor, denn `s[ i ]` wäre eine gültige Eingabe.
- OQL erlaubt es, mit den Konstruktoren Listen, Arrays, Bags, Sets und fachliche Objekte zu erzeugen. Nirgends wird erwähnt, ob diese erzeugten Objekte persistent oder transient gespeichert werden.

---

<sup>1</sup> siehe ([Catt98] S. 117)

<sup>2</sup> siehe ([Catt98] S. 95)

- Die Konstruktoren List, Bag, Set und Array können eine leere Kollektion erzeugen. Doch was kann mit einer leeren Kollektionen gemacht werden? Wie werden leere Listen im Typsystem behandelt? Im Typsystem muß von jeder Kollektion der Typ seiner Elemente bekannt sein. Wird die leere Liste als Liste über Elementen vom Typ `nil` behandelt, so wäre die leere Liste nur mit leeren Listen kompatibel und somit sinnlos. Wird aber die leere Liste als Liste gehandelt, die mit beliebigen anderen Listen kompatibel ist, so könnte die leere Liste dazu verwendet werden, um zu testen, ob eine Liste leer ist, und hätte somit wenigstens eine Funktion. Um zu prüfen, ob eine Kollektion leer ist, existiert aber schon die Funktion `exists`. Es stellt sich daher wirklich die Frage, ob die Konstruktion von leeren Kollektionen überflüssig ist und nur die Sprache aufbläht. In jedem Falle wäre aber eine Beschreibung der Semantik vonnöten. Auch ist die Frage bei der Kompatibilität von leeren Listen ungeklärt.
- Damit die Aggregatfunktionen von OQL zu denen von SQL kompatibel sind, ist es in OQL auch erlaubt, daß die Aggregatfunktionen nicht funktional verwendet werden. In ([Catt98], S. 115) wird dies durch folgende Definition/Regel verdeutlicht:

```
select count(query)..... is equivalent to
count(select query from ....)
```

Man betrachte folgende OQL-Anfrage:

```
select count(select k.name from p.kinder as k)
from Person as p
```

Wendet man auf diese Anfrage obige Regel an, so ergibt sich folgende Anfrage:

```
count(select select k.name from p.kinder as k)
from Person as p
```

Diese Anfrage liefert als Ergebnis eine Zahl, genauer gesagt die Anzahl der gespeicherten Personen. Demgegenüber liefert die Anfrage darüber - ohne Anwendung der Aggregat-Regel - eine Menge von Zahlen, wobei jede Zahl für die Anzahl der Kinder der jeweiligen Person steht. Das bedeutet: Der Vorteil der Kompatibilität zu SQL wird durch den Nachteil, daß die Semantik der Anfrage verändert wird, zunichte gemacht.

- Die von der ODMG erläuterten Sichtbarkeitsregeln sind, so wie sie (in [Catt98], S.112f) beschrieben sind, nicht richtig bzw. unpräzise., sie stehen zur übrigen Sprachbeschreibung von OQL in Widerspruch.  
Man betrachte die folgende Anfrage (aus [Catt98], S. 104f):

```
select *
from Students as x,
     x.takes as y,
     y.taught_by as z
```

```
where z.rank = "full professor"
```

Diese Anfrage liefert (nach [Catt98]) folgendes Resultat:

```
bag<struct(x:Student, y:Section, z: Professor)>
```

Dies zeigt, daß die Variablen *x*, *y*, *z* nicht nur innerhalb der *Select From Where* Anweisung als Iteratorvariablen sichtbar sind, sondern daß diese Variablen nach außen als Komponentennamen der Verbundes sichtbar werden. Der Sichtbarkeitsbereich der Variablen als Iteratorvariablen entspricht dem von der ODMG beschriebenen Bereich. Zudem müßte von der ODMG aber noch erwähnt werden, daß die Variablen nach außen hin als Komponentennamen einen (unendlichen) Sichtbarkeitsbereich haben.

Die folgende Anfrage zeigt, wie die Variable *x* des *From*-Teils als Komponentennamen nach außen einen unbeschränkten Sichtbarkeitsbereich hat:

```
first(select *
        from Students as x,
             x.takes as y,
             y.taught_by as z
        where z.rank = "full professor"
        order by x).x
```

- Das (von [Catt98], S. 92) angegebene Beispiel verletzt die Sichtbarkeitsregeln und ist somit falsch. Es zeigt aber trotzdem die im obigen Punkt erwähnte Problematik von der Sichtbarkeit von Variablen nach innen als Iteratorvariablen und nach außen als Komponentennamen.

```
first(select street, average_salary: avg(select
e.salary
                                         from partition)
        from (select (Employee)p
                from Person p
                where "has a job" in p.activities) as e
        group by street: e.address.street
        order by average_salary
        ).street
```

Der mit Fett markierte Bezeichner ist an dieser Stelle nicht sichtbar, denn Definitionen im *Select*-Teil wirken lediglich als Komponentennamen nach außen. Korrigiert man das Beispiel, so lautet die Anfrage:

```
first(select q.street, q.average_salary
        from (select street,
                    average_salary: avg(select e.salary
                                         from partition)
                from (select (Employee)p
                        from Person p
                        where "has a job" in p.activities)
```

```
        as e
        group by street: e.address.street) as q
    order by q.average_salary
).street
```

- Die ODMG hat bei der Behandlung der Sichtbarkeitsregeln die Behandlung des `for all` - Konstrukts und des `exists in` - Konstrukts vergessen.

```
for all x in e1:e2
exists x in e1:e2
```

Bei diesen beiden Konstrukten ist der Bezeichner `x` in dem Ausdruck `e2` sowie in den eingeschachtelten Ausdrücken von `e2` sichtbar.

- Die Sichtbarkeitsregeln der ODMG geben keine Auskunft darüber, ob Verschattung erlaubt ist oder nicht:

```
select *
from   Adresse a, (select * from Person a
                  where a.name like "D*") as b
```

Die Anfrage nur dann erlaubt, wenn Verschattung zugelassen wird: Denn der Bezeichner `a` der ersten Deklaration ist in der eingeschachtelten `Select From Where` - Anweisung gültig. Die zweite Deklaration von `a` (in der eingeschachtelten `Select From Where` Anweisung) würde damit die erste Deklaration von `a` überschatten.

- Die Sichtbarkeitsregeln der ODMG sind - wie vorher bereits erwähnt - nicht vollständig. Dies zeigt sich auch bei der Verwendung des `Group`-Teils:

```
select  a.strasse
from    Adresse a
group by Ort: a.ort
```

Obige Anfrage gruppiert die Adressen nach dem Ort. Eine nach der Gruppierung stattfindende Projektion auf die Straßen ist nicht möglich, da die Gruppierung nicht eine, sondern im allgemeinen viele Straßen enthält. Bei Verwendung des `Group`-Teils dürfen im `Select`-Teil nur noch die gebundenen Bezeichner sichtbar sein, für die eine Gruppierung existiert!

## 2.4. Das Java-Language Binding

Der Java-Language Binding definiert die Schnittstelle zwischen dem ODMG Object Model und der Programmiersprache Java. Das Java-Language Binding definiert u.a. die Klasse `OQLQuery`. Diese Klasse liefert die Schnittstelle für OQL-Anfragen innerhalb der Programmiersprache Java. Zudem werden die Schnittstellen `DCollection`, `DBag`, `DSet`, `DArray` und `DList` definiert. Sie geben an, wie in OQL zu verwendende Kollektionen aussehen müssen.

Eine detailliertere Beschreibung des Java-Language Bindings findet sich in dem Buch [Catt98] S.229-244).

## 3. SQL

SQL ist eine für RDBS konzipierte Sprache, die von vielen der heute kommerziell verfügbaren Datenbanksystemen implementiert wird. Im Gegensatz zu OQL ist sie nicht nur eine reine Anfragesprache, weil mit ihr zu den Anfrage-Möglichkeiten auch Relationenschemata definiert, Updates und weitere Datenmanipulationen durchgeführt werden können.

Warum wird für diese Arbeit SQL benötigt? Ziel dieser Diplomarbeit ist die Umsetzung von OQL-Anfragen an relationale Datenbanksysteme unter Verwendung eines Persistenzframeworks. Bei den heute verfügbaren kommerziellen RDBS ist ein Zugriff auf die Datenbank nur unter Verwendung der Sprache SQL möglich. Daher muß für die Umsetzung von OQL-Anfragen die Sprache SQL verwendet werden.

In dem nächsten Kapitel wird das Persistenzframework POLAR<sup>®</sup> vorgestellt, das mit nahezu jedem beliebigen RDBS betrieben werden kann. Betrachtet man die am Markt verfügbaren RDBS, so stellt man fest, daß keine generelle Kompatibilität bezgl. SQL-Anfragen zwischen den einzelnen RDBS besteht, obwohl SQL standardisiert ist. Dies liegt daran, daß entweder unterschiedliche Standards von SQL unterstützt werden (SQL86, SQL89, SQL-2, SQL-3), die meist nur teilweise implementiert sind oder daß der Hersteller SQL um bestimmte Konzepte eigenmächtig erweitert hat.

In diesem Kapitel wird dargestellt, welche Teilmenge von SQL allen RDBS, die von POLAR<sup>®</sup> unterstützt werden, gemeinsam ist. Diese Teilmenge wird als Grundstein benötigt, auf dem die Übersetzung von OQL- nach SQL-Anfragen aufbaut.

### 3.1. Geschichtliche Entwicklung von SQL

Die Firma IBM hat zu Beginn der 70er Jahre im Rahmen der Projekte SEQUEL-XRM und System-R die Entwicklung der Sprache SQL begonnen. Im Jahre 1981 hat IBM das erste kommerziell verfügbare System SQL/Data System auf den Markt gebracht, welches diese Sprache (SQL) unterstützte. In den laufenden Jahren wurde SQL für alle anderen Datenbanksysteme von IBM übernommen (vgl. [Vossen]).

Zu Beginn der 80er Jahre erkannte dann das American National Standards Institute (ANSI) die Bedeutung von SQL als universelle Datenbankabfragesprache für relationale DBS. Im Jahre 1986 wurde dann der erste SQL Standard verabschiedet, der unter dem Namen SQL-1 bzw. SQL86 bekannt wurde. Erste Ergänzungen zu diesem Standard erschienen im sog. Addendum-1. Der ab dieser Zeit gültige Standard wird dann unter dem Namen SQL89 bekannt. Weitere Ergänzungen und Modifikationen des Standards wurden 1992 im Rahmen von SQL-2 (auch unter SQL-92 bekannt, siehe [ISO]) vollzogen. Dieser Standard kann wiederum in drei verschiedenen Graden unterstützt werden - man unterscheidet in *Full SQL*, *Intermediate SQL* und *Entry SQL*.

Derzeit wurden bereits einige Vorab-Versionen des SQL-3 Standards veröffentlicht. Als Erweiterung enthält dieser Standard unter anderem die Möglichkeit, rekursive Anfragen zu formulieren (dargestellt in [Heum]).

## 3.2. Teilmenge von SQL

Zu den von POLAR<sup>®</sup> unterstützten RDBS gehört unter anderem MS-Access. Dieses RDBS unterstützt nicht einmal vollständig den SQL-89 Standard, geschweige denn den SQL-92 (Entry SQL)-Level. Das bedeutet, fast alle Konzepte von SQL, die in dem SQL-89 Standard noch nicht vorgesehen waren, können in der weiteren Arbeit nicht verwendet werden.

Es wird jetzt grob angegeben, welche Konzepte von SQL bei der Umsetzung von OQL-Anfragen verwendet werden können, dabei werden nur die Anfrage-Möglichkeiten von SQL betrachtet.

Für eine genaue Beschreibung des SQL-89 Standards - der sehr ähnlich ist mit der hier untersuchten Teilmenge von SQL - wird auf das Buch von C.J. Date - „*The SQL Standard*“ [Date89] verwiesen. Dort findet sich auch eine Beschreibung der Syntax in Backus-Naur-Form. Eine Beschreibung des SQL-92 Standards, von dem hier fast kein zusätzliches Konzept verwendet werden kann, findet sich in dem zugehörigen ISO/IEC 9075:1992(E) Standard [ISO].

### 3.2.1. Select From Where - Anweisung

Jede SQL-Anfrage besteht im wesentlichen aus einer Select From Where Anfrage. Diese dient zum Auswählen der gewünschten Informationen aus der Datenbank.

Der Select-Teil einer solchen Anweisung gibt eine Liste von Ausdrücken (s.u.) an, welche die vorzunehmende Projektion beschreibt, d.h. es wird angegeben, welche Spalten die Ergebnistabelle haben soll und wie sich die Daten der Spalten zusammensetzen. Wird anstelle der Liste von Ausdrücken in dem Select-Teil das Zeichen \* verwendet, so sollen alle Spalten der in dem From-Teil angegebenen Tabellen dargestellt werden, in diesem Fall müssen die Spalten der Ergebnistabelle nicht explizit aufgezählt werden.

Generell werden von SQL keine Duplikatzeilen aus dem Anfrageergebnis entfernt. Sollen Duplikatzeilen entfernt werden, so kann man dies durch Angabe des Schlüsselwortes DISTINCT vor der Liste von Ausdrücken im Select-Teil erreichen.

Die Tabellen, aus denen die Daten ermittelt werden sollen, werden in dem From-Teil angegeben. Hier kann zusätzlich zu jedem Tabellennamen ein Aliasname vereinbart werden, der dann für die Formulierung von Ausdrücken verwendet werden kann. Es sei ausdrücklich darauf hingewiesen, daß in der hier betrachteten Teilmenge von SQL nur Tabellennamen eventuell mit Aliasnamen im From-Teil verwendet werden dürfen. Somit können keine Select

From Where-Anweisungen in den From-Teil einer Select From Where-Anweisung eingeschachtelt werden (im Gegensatz dazu [ISO], S. 143)

Werden in dem From-Teil einer Select From Where-Anweisung mehrere Tabellen angegeben, so werden die Tabellen miteinander verbunden - d.h. es wird das kartesische Produkt aus allen Mengen die im From-Teil angegeben wurden (die ja in Form von Tabellen vorliegen) gebildet.

Die Auswahl der Zeilen der Ergebnistabelle erfolgt durch die Auswertung der Suchbedingung (s.u.) in dem Where-Teil. Die Suchbedingung kann selbst wieder Select From Where - Anweisungen enthalten. Der Where-Teil ist optional, wird er weggelassen, so werden alle Zeilen ausgewählt.

Beispiele:

```
SELECT a.strasse
FROM   Adresse as a
WHERE  a.ort LIKE 'Entenhausen'
```

liefert alle Straßen der Stadt Entenhausen.

```
SELECT *
FROM   Adresse as a, Person as p
```

liefert das kartesische Produkt aus den Mengen Adressen und Personen.

Die Auswertung einer Select From Where Anweisung läuft grundsätzlich in folgenden drei Schritten ab:

1. Zuerst wird das kartesische Produkt aus den in dem From-Teil angegebenen Operanden-Tabellen gebildet.
2. Für die erzeugten Tupel wird der Where-Teil ausgewertet, der als Selektionsbedingung dient.
3. Das erzeugte Zwischenergebnis wird auf die in dem Select-Teil angegebenen Attribute bzw. Werte projiziert.

#### **3.2.2. Ausdrücke**

Ein Ausdruck kann durch die Verknüpfung von Spaltennamen, Konstanten und Aggregatfunktionen gebildet werden. Zulässige Verknüpfungen sind die Operationen Addition, Multiplikation, Division, Subtraktion und Modulo, jedoch nur soweit dies mit den jeweiligen Typen vereinbar ist. Durch Klammersetzung können die für die Operationen gültigen Operator-Prioritäten geändert werden.

### 3.2.3. Suchbedingungen

Eine Suchbedingung gibt eine Bedingung an, die entweder den Wert true, false oder NULL annehmen kann. Im allgemeinen ist eine Suchbedingung eine logische Verknüpfung von Prädikaten - dabei sind die Operatoren AND, OR und NOT für die Verknüpfung der Prädikate definiert.

### 3.2.4. Prädikate

Prädikate können u.a. mit den binären Vergleichsoperatoren <, <=, >, >= und = angewandt auf Ausdrücke erstellt werden. Ist einer der Operanden des Vergleichs NULL, so ist der Wert des Vergleichs NULL, andernfalls entweder true oder false.

Außer mit den Vergleichsoperatoren können Prädikate mit den folgenden Konstrukten gebildet werden:

Operator	Bedeutung
LIKE	Mit dem LIKE-Operator können Zeichenketten auf bestimmte Muster geprüft werden. Ein Muster ist dabei eine Zeichenkette, die als Wildcard-Zeichen den Unterstrich und das Prozentzeichen enthalten kann. Der Unterstrich steht für genau ein beliebiges Zeichen, das Prozentzeichen für eine beliebige Zeichenkette, die auch leer sein kann. Ein ESCAPE-Zeichen, das die Bedeutung der Wildcard-Zeichen aufhebt, existiert in der hier untersuchten Teilmenge von SQL nicht!
EXISTS	Mit der EXISTS-Funktion kann geprüft werden, ob eine Tabelle Zeilen enthält oder nicht. Dabei stellt das Argument dieser Funktion eine Select From Where Anweisung dar.
IN	Mit Hilfe des IN-Operators kann geprüft werden, ob ein Ausdruck in einer Menge von Werten vorkommt.
BETWEEN	Mit dem BETWEEN-Konstrukt kann geprüft werden, ob ein Wert innerhalb vorgegebener Grenzen liegt.

Die Vergleichsoperatoren <, >, <=, >= und = können mit den Quantoren ALL und ANY verwendet werden. ALL bedeutet, daß das Prädikat für alle Zeilen gelten muß, ANY bedeutet, daß der Vergleich für mindestens eine Zeile gelten muß.

Beispiel:

```
SELECT *
FROM   Person
WHERE  name = ANY (SELECT name FROM Mitarbeiter)
```

liefert alle Personen, die Mitarbeiter sind.

#### 3.2.5. Group by und Having

Mit Hilfe des Group by-Teils können bestimmte Tupel einer Tabelle anhand gemeinsamer Werte gruppiert werden. Jedes Attribut, das in dem Group-by-Teil vorkommt, muß ein Name einer Spalte einer im From-Teil verwendeten Tabelle sein. Die Ergebnistabelle die unter Verwendung von Group-by resultiert, wird folgendermaßen erstellt: Zunächst wird eine Ergebnistabelle aufgrund des From- und Where-Teils erzeugt. Diese wird durch den Group-by-Teil verändert. Dabei werden die Zeilen derart gruppiert, daß in einer Gruppe alle Zeilen zusammengefaßt sind, die in den durch den Group-by-Teil vereinbarten Spalten die gleichen Werte aufweisen. Der Select-Teil muß dann für jede Gruppe genau einen Wert erzeugen.

Hinweis: Das Ergebnis der Auswertung einer Group-by Anweisung ist keine Tabelle - die Überführung in eine zulässige Tabellenform geschieht erst durch den Select-Teil.

Der Having-Teil bildet eine Auswahlbedingung für die Gruppen, er ist somit für Gruppen das, was der Where-Teil für Zeilen darstellt.

Beispiel für die Verwendung von Group by und Having:

```
SELECT *
      FROM Adresse
GROUP BY ort
HAVING COUNT(ort)>2
```

#### 3.2.6. Order by

Die Reihenfolge, in der die Tupel in der Tabelle vorkommen, ist üblicherweise nicht spezifiziert - also implementierungsabhängig. Dies liegt daran, daß das Ergebnis einer Anfrage eine Menge bzw. eine Multimenge ist. Auf diesen beiden Datentypen ist keine Reihenfolge der Elemente definiert. Durch die Order by Anweisung kann eine bestimmte Reihenfolge der Tupel in der Ergebnis-Tabelle erzwungen werden. Durch Angabe des Schlüsselwortes ASC kann aufsteigend, durch das Schlüsselwort DESC kann absteigend sortiert werden.

Dabei kann die Sortierichtung (aufsteigend oder absteigend) für jede Spalte der Ergebnistabelle unterschiedlich definiert werden. Eine Sortierung ist nur nach Spalten möglich, nicht nach beliebig definierbaren Sortierkriterien!

Beispiel:

```
SELECT *  
FROM Adresse  
ORDER BY a.ort, a.strasse DESC, a.postleitzahl ASC
```

liefert die Adressen aufsteigend sortiert nach Orten. Gleiche Orte werden absteigend nach Straßen definiert. Sind sowohl der Ort und die Straße gleich, so wird aufsteigend nach der Postleitzahl sortiert.

#### 3.2.7. Union

Mehrere gleichstrukturierte Ergebnistabellen können mit Hilfe des binären Operators UNION zu einer einzigen Ergebnistabelle zusammengefaßt werden. Dabei sind Tabellen genau dann gleichstrukturiert, wenn die Reihenfolge der Spalten und deren Datentypen übereinstimmen.

Im Gegensatz zu einer einfachen Select From Where Anweisung, die Duplikate nur dann entfernt, wenn das Schlüsselwort DISTINCT angegeben wurde, werden in einer mit UNION erzeugten Ergebnistabelle Duplikatzeilen automatisch entfernt. Wird jedoch das Schlüsselwort ALL angegeben, so bleiben Duplikate erhalten.

#### 3.2.8. Aggregatfunktionen

SQL unterstützt genauso wie OQL fünf eingebaute Aggregatfunktionen: COUNT, SUM, AVG, MIN und MAX. Die Aggregatfunktion COUNT dient zum Zählen der Zeilen einer Tabelle oder einer Spalte, MAX bestimmt das Maximum einer numerischen oder alphanumerischen Spalte, MIN das Minimum. Die Funktion SUM liefert die Summe der Werte einer numerischen Spalte und AVG bestimmt den Mittelwert einer numerischen Spalte.

Im Gegensatz zu OQL werden die Aggregatfunktionen in SQL in einer nicht-funktionalen Schreibweise verwendet (vgl. hierzu Kapitel 2.3.3)

Beispiele:

```
SELECT COUNT(*) FROM Adresse  
SELECT COUNT(ort) FROM Adresse  
SELECT SUM(gehalt) FROM Mitarbeiter
```

Bis auf den Aufruf von COUNT(\*) werden alle NULL-Werte vor der Auswertung der jeweiligen Aggregatfunktion eliminiert. In dieser Teilmenge von SQL kann das Schlüsselwort DISTINCT nicht in Verbindung mit einer Aggregatfunktion verwendet werden. Es darf auch nur eine Aggregatfunktion in dem Select-Teil einer Select From Where Anweisung vorkommen.

Somit sind folgende Anfragen im weiteren *unzulässig*, obwohl sie ansonsten von einigen RDBS zugelassen werden:

```
SELECT DISTINCT COUNT(*) FROM Adresse
SELECT COUNT(DISTINCT ort) FROM Adresse
SELECT COUNT(ort), COUNT(strasse) FROM Adresse
```

Wird in dem Select-Teil eine Aggregatfunktion benutzt, so wird für diese Spalte nur ein einziger Wert berechnet. Eine Kombination mit anderen Spalten, auf die keine Spaltenfunktion angewendet werden soll, ist deshalb prinzipiell *unzulässig*:

```
SELECT m.name, MAX(m.gehalt)
FROM Mitarbeiter m
```

#### 3.2.8. Abschließende Bemerkungen

Es sei hier nochmals bemerkt, daß die hier untersuchte Teilmenge von SQL die folgenden Konstrukte nicht unterstützt. Dies liegt daran, daß ein oder mehrere RDBS (größtenteils MS-Access 7.0) dieses Konstrukt nicht implementieren:

- Im Gegensatz zu SQL-92 kann keine Select From Where Anweisung in dem From-Teil einer anderen Select From Where-Anweisung geschachtelt werden (siehe dazu [ISO] Seite 143 i.V.m Seite 95 und 165).
- Das Schlüsselwort DISTINCT kann nicht in Verbindung mit den Aggregatfunktionen verwendet werden, was im Gegensatz zur SQL-92 Definition steht (siehe dazu [ISO] Seite 98).
- Es kann immer nur eine Aggregatfunktion in einem Select-Teil vorkommen, genau wie in der SQL-92 Definition ([ISO] Seite 98).
- Es gibt im Gegensatz zu SQL-92 keine SQL-Unterstützung für den Mengen-Durchschnitt und die Mengen-Differenz. In SQL-92 gibt es eine Unterstützung für den Mengen-Durchschnitt INTERSECT und die Mengen-Differenz EXCEPT (siehe dazu [ISO], Seite 159).
- Die UNIQUE-Funktion wird in der hier untersuchten Teilmenge von SQL nicht unterstützt. Demgegenüber unterstützt SQL-92 diese Funktion (siehe [ISO], Seite 226).
- Es gibt keine Operationen für die Manipulation von Zeichenketten. Demgegenüber gibt es in SQL-92 Operationen für die Manipulation von Zeichenketten (siehe [ISO], Seite 298ff).

# 4. POLAR

Viele Anwendungen werden heutzutage mit objektorientierten Programmiersprachen entwickelt. Da einerseits auf dem Markt hauptsächlich relationale Datenbanksysteme existieren, und andererseits die Kosten für die Umwandlung von bereits existierenden relationalen Datenbankmodellen auf objektorientierte Datenbankmodelle sehr hoch sind, werden oft objektorientierte Programmiersprachen in Verbindung mit dem relationalen Datenmodell verwendet. Dies führt zu dem Problem des *impedance mismatch*, d.h. die innerhalb des gleichen Systems zu verwendenden Datenmodelle sind teilweise nicht verträglich.

Um dieses Problem zu umgehen, hat die Firma IBL Ingenieurbüro Letters GmbH ein Produkt mit Namen POLAR<sup>®</sup>, was für *Persistent Object Language Above Relations* steht, entwickelt.

[Wan94] schreibt über POLAR<sup>®</sup>: "*Das objektorientierte Datenbankmodell POLAR<sup>®</sup> erweitert das Datenmodell der objektorientierten Programmiersprache so, daß eine Ebene zwischen relationaler Datenbank und Programmiersprache generiert werden kann, welche die Konfliktsituation der verschiedenen Datenmodelle behebt bzw. minimiert.*"

POLAR<sup>®</sup> erweitert objektorientierte Programmiersprachen um ein objektorientiertes Datenbankmodell. Zur Speicherung der Objekte wird ein relationales Datenbanksystem verwendet, d.h. POLAR<sup>®</sup> setzt auf ein bereits bestehendes relationales Datenbanksystem auf, indem es u.a. Mechanismen für die Abbildung von Klassen auf relationale Strukturen anbietet. Von dem Framework werden die Programmiersprachen Java, C++ und SmallTalk unterstützt. Für die Datenbanksysteme Sybase SQL-Server, Microsoft SQL-Server, Oracle, Informix SE, IBM DB/2, Gupta SQLBase, MS-Access und für beliebige andere Datenbanksysteme (über ODBC) existieren Anbindungen.

## 4.1. Systemarchitektur

POLAR<sup>®</sup> ist ein Persistenzframework, welches als zusätzliche Schicht zwischen objektorientierte Anwendungsentwicklung und relationale Datenbank geschoben wird. Die interne Systemarchitektur dieses Frameworks läßt sich in die drei folgenden Schichten aufteilen:

1. Modell-Ebene (Interne Schicht)
2. Zugriffs-Ebene
3. Datenbank-Ebene

Zusätzlich zu diesen drei Ebenen existieren noch eine Reihe von Werkzeugen für die Anwendungsentwicklung.

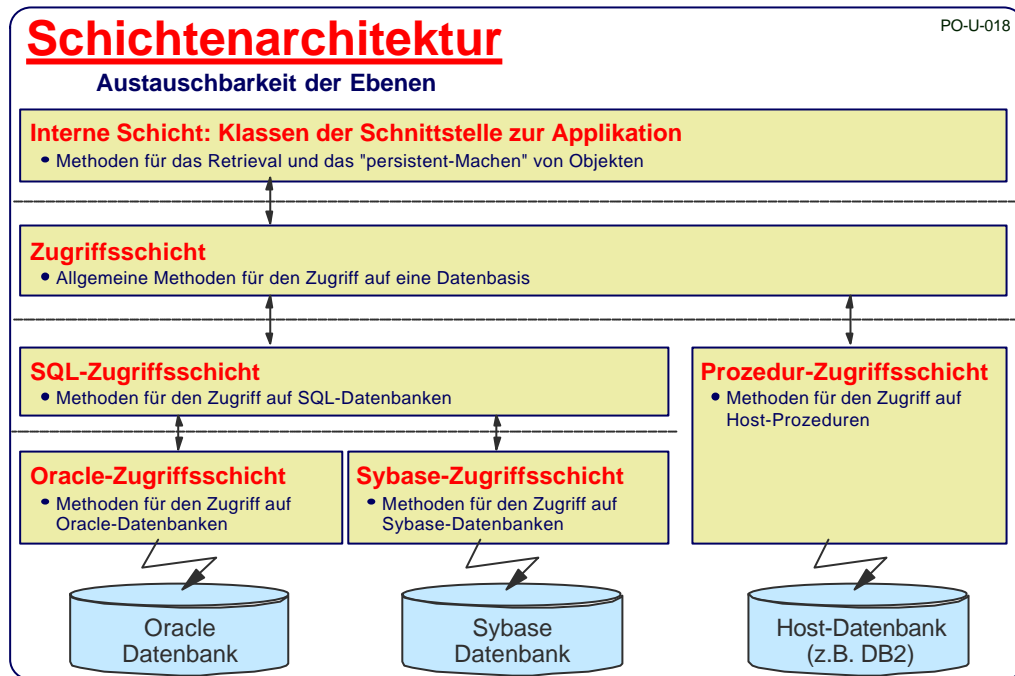


Abbildung 1: Systemarchitektur von POLAR (aus [IBL97], S. 64)

1. *Modell-Ebene bzw. Interne Schicht:* Sie enthält u.a. die Klasse PObject, die als Oberklasse für alle fachlichen Klassen dient, deren Objekte potentiell persistent gemacht werden sollen. Die Modell-Ebene stellt die Schnittstelle dar, die dem Entwickler beim Einsatz des POLAR<sup>®</sup> Datenbankmodells zur Verfügung steht.
2. *Zugriffs-Ebene:* In der Zugriffs-Ebene wird die Umwandlung von Anfragen aus den Klassen der Modell-Ebene in Anfragen des speziellen Zieldatenbanksystems vorgenommen. Ebenso werden die Ergebnisse aus dem Datenbanksystem in eine für die Modellebene spezifizierte Form überführt.
3. *Datenbank-Ebene:* Die Datenbank-Ebene stellt die Schnittstelle zum speziellen Datenbanksystem her. Diese Ebene muß u.a. das Absetzen von SQL-Befehlen an das Datenbanksystem ermöglichen.

Durch die Aufteilung in unabhängige Schichten wird erreicht, daß jedes beliebige relationale Datenbanksystem mit POLAR<sup>®</sup> gekoppelt werden kann. Dazu muß nur die Zugriffsebene und die Datenbankebene ausgetauscht werden. Die Modellebene, als einzige für den Benutzer sichtbare Schicht, bleibt davon unberührt.

## 4.2. Abbildung des OO-Datenmodells in dem RDBS

Der folgende Abschnitt beschreibt wie POLAR<sup>®</sup> die statische Koppelung von objektorientiertem Datenbankmodell zu relationalen Datenbankmodell vornimmt, d.h. es wird erläutert, wie ein objektorientiertes Datenbankmodell in einem relationalen Datenbankmodell abgebildet werden kann, und welche Informationen dafür notwendig sind.

Prinzipiell wird jedes Attribut einer Klasse in genau einer Spalte einer Datenbanktabelle abgelegt. Zudem enthält jede Klasse ein Attribut für die Aufnahme der Objektidentität (OID).

### 4.2.1. Abbildung der Klassenhierarchie

In POLAR<sup>®</sup> gibt es drei verschiedene Möglichkeiten für die Abbildung einer Klassenhierarchie in den Tabellen einer relationalen Datenbank:

1. *Daten jeder Klasse in einer eigenen Tabelle:*

Jede Klasse innerhalb einer Klassenhierarchie ist genau einer Tabelle zugeordnet. Diese Tabelle enthält die Attribute der Klasse als Tabellenspalten. Der Nachteil dieser Methode ist, daß die Daten eines Objektes auf Grund der Vererbungs-Eigenschaft sich auf mehrere Tabellen verteilen können, womit zur Bearbeitung und Veränderung von Objekten mehrere Tabellen bearbeitet werden müssen, was zu Performance-Verschlechterungen führen kann. Der Vorteil dieser Methode ist der geringe Platzverbrauch, da in keinem Tupel der Tabelle leere Elemente enthalten sind.

2. *Nur für Unterklassen wird eine Tabelle angelegt:*

Nur für die Unterklassen (Blätter in der Klassenhierarchie) werden Tabellen angelegt, welche die Attribute der Klassen als Spalten enthalten. Der Vorteil dieser Methode besteht darin, daß bei Objektoperationen (anlegen, ändern und löschen) jeweils nur eine Tabelle bearbeitet werden muß. Nachteil dieser Methode ist, daß für Objekte, die keine Blätter in der Klassenhierarchie sind, einige Spalten leer bleiben, was zu einem größeren Platzverbrauch als bei Methode 1 führt.

3. *Nur für die Oberklasse wird eine Tabelle angelegt:*

Nur für die Oberklasse einer Vererbungshierarchie wird eine Tabelle angelegt, in der die Attribute dieser Klasse und aller Subklassen als Tabellenspalten repräsentiert werden. Alle Unterklassen teilen sich diese eine Tabelle. Nachteil dieser Methode ist, daß es sehr große Tabellen mit sehr vielen Spalten geben kann. Außerdem wird sehr viel Platz benötigt, da viele Attribute von der jeweiligen Instanz gar nicht verwendet werden. Ein weitere Nachteil dieser Methode ist, daß bei einer Reihe von kommerziell verfügbaren RDBS die Tupellänge beschränkt ist. Der Vorteil dieser Methode ist die mit Abstand beste Performance. So ist, wenn die Klasse der erwarteten Objekte noch nicht feststeht nur eine Tabelle zu durchsuchen.

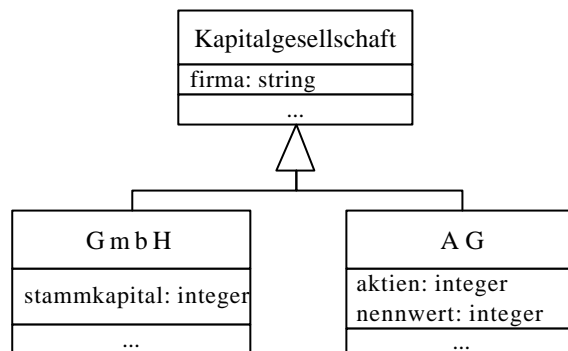
#### 4. POLAR

---

POLAR<sup>®</sup> erlaubt die Verwendung aller drei Alternativen, so daß je nach Einsatzgebiet die beste Alternative ausgewählt werden kann. Auch das Mischen der Alternativen ist möglich, wenn drei oder mehr Klassen in einer hierarchischen Beziehung stehen. (siehe dazu [IBL97], Seite 30f)

Mehrfachvererbung ist in POLAR<sup>®</sup> nicht vorgesehen, da diese in den Programmiersprachen Java und SmallTalk auch nicht möglich ist.

*Objektmodell:*



*Daten jeder Klasse in einer eigenen Tabelle:*

OID	FIRMA
1##1##2	Entenhausen AG
1##1##3	Duck GmbH
1##1##4	DD Stiftung

Tabelle für Klasse Kapitalgesellschaft

OID	AKTIEN	NENNWERT
1##1##2	500 000	50 DM

Tabelle für Klasse AG

OID	STAMMKAPITAL
1##1##3	100 000 DM

Tabelle für Klasse GmbH

*Nur für die Unterklassen wird eine Tabelle angelegt:*

OID	AKTIEN	NENNWERT	FIRMA
1##1##2	500 000	50 DM	Entenhausen AG
1##1##4	NULL	NULL	DD Stiftung

Tabelle für Klasse AG

OID	STAMMKAPITAL	FIRMA
1##1##3	100 000 DM	Duck GmbH

Tabelle für Klasse GmbH

Nur für die Oberklasse wird eine Tabelle angelegt:

OID	AKTIEN	NENNWERT	STAMMKAPITAL	FIRMA
1##1##2	500 000	50 DM	NULL	Entenhausen AG
1##1##3	NULL	NULL	100 000 DM	Duck GmbH
1##1##4	NULL	NULL	NULL	DD Stiftung

Tabelle für Klasse Kapitalgesellschaft

Abbildung 2: Auflösen der Klassenhierarchie im RDBS

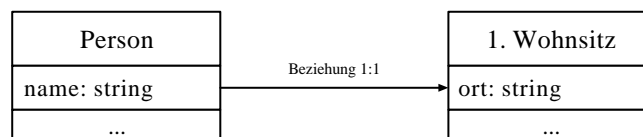
## 4.2.2. Abbildung von Beziehungen

Generell wird bei den nachfolgenden Beziehungen davon ausgegangen, daß jede Klasse ein Attribut für die Aufnahme der Objektidentität (OID) enthält. Die Objektidentität dient als Primärschlüssel für das jeweilige Objekt der Klasse. Für die Abbildung von Beziehungen wird die Objektidentität (OID) verwendet, indem Fremdschlüssel definiert werden, welche die OID aufnehmen.

### 4.2.2.1. Abbildung von 1:1-Beziehungen

Bei 1:1-Beziehungen muß ein zusätzliches Attribut (eine zusätzliche Spalte im RDMS) für die Aufnahme des Fremdschlüssels angelegt werden. Dabei kann der Fremdschlüssel sowohl in der Quell- als auch in der Zielklasse vorkommen. Soll eine Beziehung zwischen zwei Objekten der beiden Klassen besonders schnell zu ermitteln ein, so kann ein Fremdschlüssel auch in beiden Klassen angelegt werden.

Objektmodell:



1. Möglichkeit, Fremdschlüssel in der Quellklasse:

OID	NAME	WOHNSITZ-ID	OID	ORT
1##1##1	Duck	1##1##8	1##1##8	Entenhausen
1##5##1	Gans	1##2##7	1##2##7	Entenhausen
...	...	...	...	...

Tabelle für Klasse Person
Tabelle für Klasse 1. Wohnsitz

2. Möglichkeit, Fremdschlüssel in der Zielklasse:

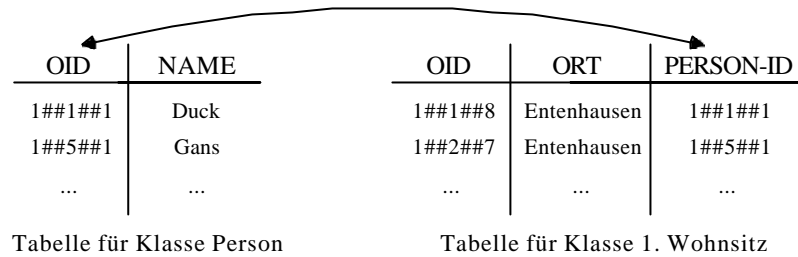


Abbildung 3: Abbildung von 1:1-Beziehungen im RDBS

4.2.2.1. Abbildung von 1:n-Beziehungen

Wie bei den 1:1-Beziehungen muß bei 1:n-Beziehungen ein zusätzliches Attribut für die Aufnahme des Fremdschlüssels angelegt werden. Im Gegensatz zu den 1:1-Beziehungen kann jedoch die Klasse, die den Fremdschlüssel aufnehmen soll, nicht mehr frei gewählt werden. Der Fremdschlüssel muß in der Klasse, die mit der Kardinalität  $n$  an der Beziehung beteiligt ist, abgelegt werden. Eine Ablage des Fremdschlüssels in der Klasse, die mit der Kardinalität  $1$  an der Beziehung beteiligt ist, ist nicht möglich, da bei RDBS alle Tupelkomponenten in 1. Normalform sein müssen. Um den Fremdschlüssel in der Klasse, die mit der Kardinalität  $1$  an der Beziehung beteiligt ist, ablegen zu können, wären geschachtelte Relationen notwendig (NF<sup>2</sup>-Modell).

Objektmodell:



Fremdschlüssel in der Klasse mit Kardinalität  $n$ :

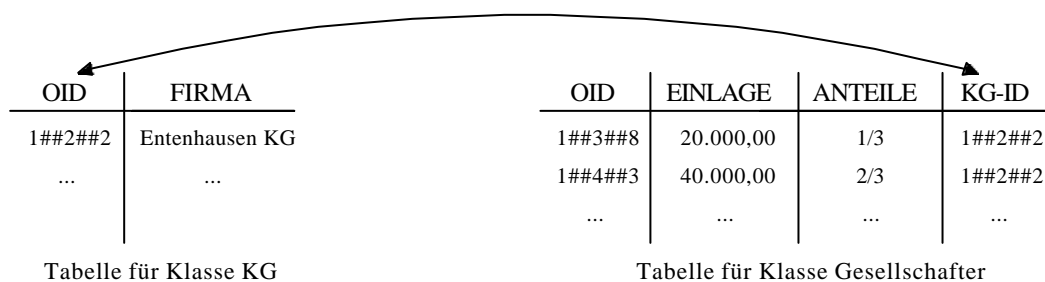


Abbildung 4: Abbildung von 1:n-Beziehungen im RDBS

### 4.3. Schema - Aufbau

Im letzten Abschnitt wurde grob beschrieben, wie ein objektorientiertes Datenbankmodell in ein relationales Datenbankmodell abgebildet werden kann. Für die automatische Erstellung eines relationalen Datenbankschemas aus einem objektorientierten Datenbankmodell gibt es mehrere Möglichkeiten: So können IDL-Beschreibungen, SQL-Data-Definition Befehle oder Objektmodellierungen als Grundlage für die Erstellung des Datenmodells verwendet werden. Aus diesem Modell kann mit Hilfe eines Import-Werkzeugs das Metamodell (Objektmodell mit Informationen über die relationale Abbildung) der Datenbank für ein vorher bestimmtes relationales Datenbanksystem erstellt werden. Damit kann automatisch Java-Programmcode für die Klassen erzeugt werden, welcher durch den Entwickler um die notwendige Funktionalität ergänzt werden kann. Außerdem existiert ein Tool mit dem automatisch die Dokumentation des Objektmodells erstellt werden kann.

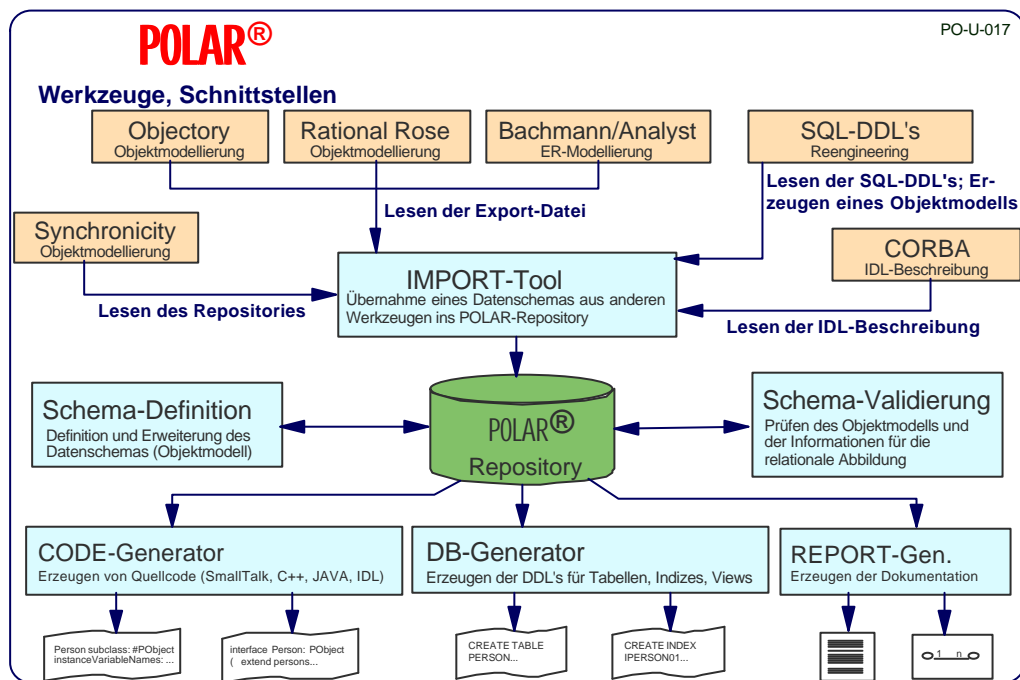


Abbildung 5: Übersicht über die Werkzeuge und Schnittstellen (aus [IBL97], S. 67)

Der Aufbau eines vollständigen Datenmodells inklusive dem Erstellen eines zugehörigen Datenbankschemas teilt sich üblicherweise in folgende vier Schritte auf: Import eines Datenmodells, Nacharbeiten des Datenmodells, Hinzufügen von Abbildungsinformationen und Optimierungen am Modell.

## 4.4. Der POLAR-Kern

Der POLAR<sup>®</sup>-Kern besteht aus dem Persistenzmodell POS (Persistent Object Service) das zur Aufgabe hat, die „relationale Welt“, also die eingesetzten relationalen Datenbanksysteme mit der „objektorientierten Welt“ - also Java - zu koppeln.

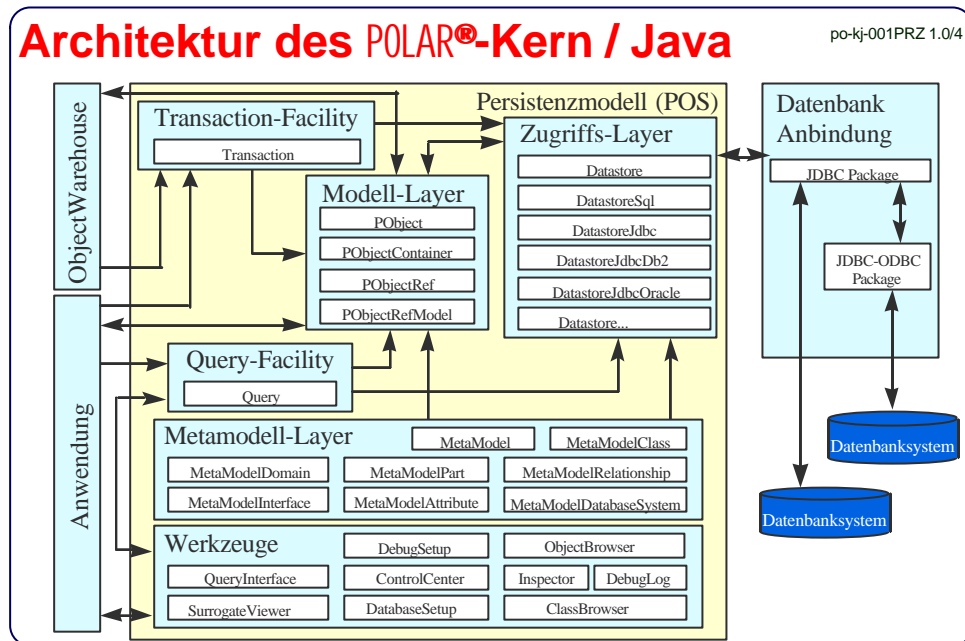


Abbildung 6: Architektur des POLAR-Kerns (aus [IBL99], S. 10)

Obige Abbildung zeigt die Komponenten, aus denen sich das Persistenzmodell POS zusammensetzt:

- Der *Metamodell-Layer* enthält alle Informationen über das Klassenmodell und dessen Abbildung auf das RDBS.
- Der *Modell-Layer* enthält u.a. die Oberklasse *PObject*, von der alle fachlichen Klassen abgeleitet werden müssen.
- Der *Zugriffslayer* ermöglicht die Entkopplung des Persistenzmodells vom tatsächlich verwendeten relationalen Datenbanksystem.
- Die *Transaction-Facility* enthält die Klassen für das Transaktionsmanagement.
- Die *Query-Facility* enthält Klassen für die Formulierung von assoziativen Objekt-Anfragen an das RDBS.

Für die Übersetzung von OQL-Anfragen in SQL-Anfragen sind die Informationen des Metamodell-Layers über das Objektmodell und dessen Abbildung auf das RDBS von zentraler Bedeutung. Einige wichtige Klassen des Metamodell-Layers, die sich in dem Package '*de.ibl.polar.pos*' befinden, werden jetzt kurz beschrieben:

Klasse	Beschreibung
<i>MetaModel</i>	Diese Klasse stellt die Schnittstelle zum aktuell geladenen Metamodell dar. Sie enthält eine Referenz auf ein Objekt der Klasse <i>MetaModelHolder</i> .
<i>MetaModelHolder</i>	Diese Klasse enthält alle Informationen über ein komplettes Modell. So kennt diese Klasse alle Klassen-, alle Beziehungs-, alle Domänen- und alle Datenbanksystem-Beschreibungen.
<i>MetaModelClass</i>	Diese Klasse beschreibt eine fachliche Klasse inklusive der relationalen Abbildung. Sie bietet u.a. eine Methode an, die den Namen der Tabelle bzw. der View angeben, aus welcher Objekte dieser Klasse rekonstruiert werden können. Auch Methoden für den Zugriff auf Primär- und Fremdschlüssel stehen zur Verfügung.
<i>MetaModelAttribute</i>	Diese Klasse beschreibt ein fachliches Attribut - inklusive relationaler Abbildung - einer Klasse.
<i>MetaModelRelationship</i>	Diese Klasse beschreibt eine fachliche Beziehung - inklusive relationaler Abbildung - zwischen zwei Klassen.
<i>MetaModelDomain</i>	Diese Klasse beschreibt eine Domäne mit den Informationen, die zur Verwendung in der Programmier-sprache notwendig sind.

Für das Wiederfinden von Objekten im persistenten Speicher bietet POLAR<sup>®</sup> bisher zwei Möglichkeiten an:

1. *Query by Example*: Es werden ein oder mehrere Objekte als eine Art Muster für die Suche angegeben. Das System sucht dann all diejenigen Objekte, die mit dem Muster übereinstimmen. Hat ein Attribut im Muster keinen Eintrag, so wird dieses Attribut bei der Suche vernachlässigt.
2. *Suche mit SQL*: Reichen die Suchmöglichkeiten von Query by Example nicht aus, so können SQL-Befehle auch direkt formuliert werden. Dies ist jedoch nicht unproblematisch, da SQL-Befehle nicht auf der Objekt-Ebene, sondern auf der Ebene der relationalen Abbildung formuliert werden müssen.

Da bestimmte Anfragen durch *Query by Example* nicht erzeugt werden können und die Suche mit SQL nicht auf der Ebene der Objekte stattfindet, wird in dieser Arbeit die Umsetzung von OQL-Anfragen untersucht und implementiert. Dabei wird die von POLAR<sup>®</sup> zur Verfügung gestellte direkte Suche mit SQL verwendet. Für die Erzeugung der SQL-Befehle werden die oben beschriebenen Klassen verwendet, die das aktuell geladene

#### 4. POLAR

---

Metamodell beschreiben. Alle anderen Komponenten von POLAR® sind für diese Arbeit irrelevant.

## 5. Grundlagen des Compilerbaus

Ein wesentlicher Teil dieser Arbeit besteht in der Entwicklung eines Compilers für die Umsetzung von OQL-Anfragen in adäquate SQL-Anfragen. Dafür werden zu Beginn dieses Kapitels die grundlegenden Phasen des Compilerbaus erläutert. Von diesen Grundlagen ausgehend werden die Hilfsprogramme JavaCC und JJTree beschrieben, welche zur Entwicklung eines OQL- Parsers verwendet wurden.

### 5.1. Theoretische Grundlagen

„Ein *Compiler* ist ein Programm, das ein in einer bestimmten Sprache geschriebenes Programm liest und in ein äquivalentes Programm einer anderen Sprache übersetzt.“ (aus [Aho86], S.1)

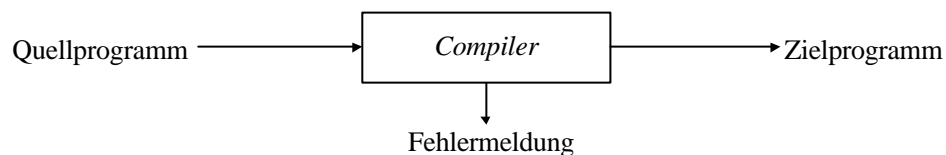


Abbildung 7: Arbeitsweise eines Compilers (aus [Aho86], S.1)

Der Übersetzungsprozeß von der Quellsprache in die Zielsprache besteht aus zwei Phasen, der *Analyse* und der *Synthese*. In der Analysephase wird das Quellprogramm welches in der Quellsprache geschrieben ist, in seine Bestandteile zerlegt und in eine bestimmte Zwischendarstellung gebracht. Die Synthesephase erzeugt das gewünschte Zielprogramm aus der in der Analysephase erzeugten Zwischendarstellung.

#### 5.1.1. Analysephase

Die Analysephase hat zum Ziel, das Quellprogramm in eine bestimmte *Zwischendarstellung* zu bringen, meist in Form eines speziellen Baumes, dem sogenannten *Syntaxbaum*. Läßt sich keine *Zwischendarstellung* erzeugen, da das Quellprogramm oder Teile davon nicht in der Quellsprache enthalten ist, so sollte mit einer aussagenden Fehlermeldung reagiert werden.

Die Analysephase besteht üblicherweise aus den folgenden drei Teilen:

1. *Lexikalische Analyse*: Die lexikalische Analyse betrachtet als Eingabe eine Folge von Zeichen, aus welchen die Wörter der Quellsprache (=Tokens) aufgebaut werden.

2. *Syntaktische Analyse*: Die syntaktische Analyse betrachtet als Eingabe eine Folge von Tokens, aus welchen die syntaktische Struktur des Programms erkannt wird. Für die weitere Verarbeitung wird diese in eine Zwischendarstellung gebracht.
3. *Semantische Analyse*: Die semantische Analyse überprüft einige notwendige Eigenschaften wie Gültigkeitsregeln, Deklariertheitseigenschaften und Typ-konsistenzen, die nicht durch eine kontextfreie Grammatik beschreibbar sind.

### 5.1.1.1. Lexikalische Analyse

Der Scanner, als Produkt der lexikalischen Analyse bildet die erste Phase des Compilers. Seine Aufgabe besteht darin, die Eingabedaten Zeichen für Zeichen zu lesen und als Ausgabe eine Folge von Symbolen (Tokens) zu erzeugen, die dann im Parser, dem Produkt der syntaktischen Analyse weiter verarbeitet werden können.

Die Arbeitsweise eines Scanners zeigt sich an folgendem Beispiel:

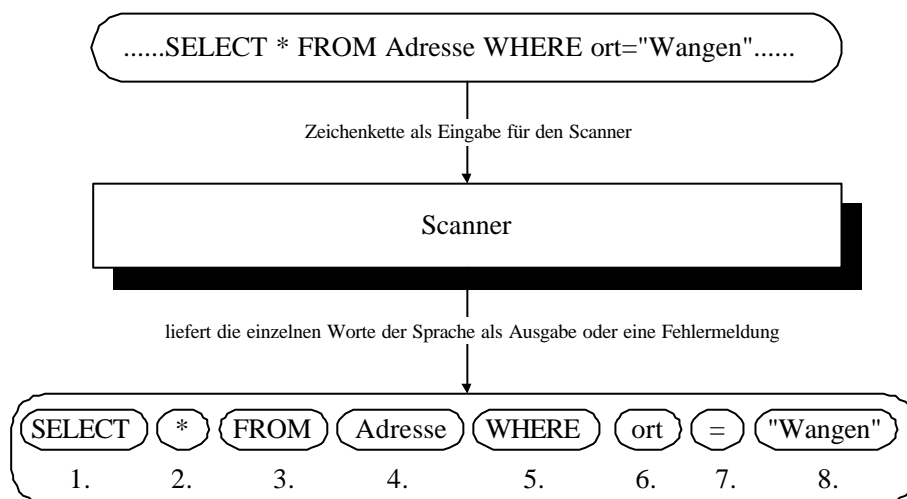


Abbildung 8: Beispiel für die Arbeitsweise eines Scanners

Da der Scanner der Teil des Compilers ist, der den Quelltext liest, werden ihm üblicherweise einige weitere Aufgaben übertragen, so zum Beispiel das Entfernen von Kommentartexten, Leerzeichen, Tabulatorzeichen und Zeilenvorschubzeichen (CR).

Eine weitere wichtige Aufgabe besteht darin, sich die Anzahl der bisherigen Zeilenwechsel oder die Startpositionen der einzelnen Tokens im Quelltext zu merken, damit im Falle eines Fehlers dem Benutzer die Position der Fehlerursache angegeben werden kann.

### 5.1.1.2. Syntaktische Analyse

Der Parser, als Produkt der syntaktischen Analyse hat zur Aufgabe, die syntaktische Struktur der Programme zu bestimmen und diese in eine geeignete interne Darstellung zu bringen. Eine mögliche Darstellungsform ist dabei der Syntaxbaum. Dieser kann dann später mit (statischer) semantischer Information angereichert werden, zu Optimierungszwecken transformiert und schließlich in eine Zielsprache übersetzt werden.

Eine weitere wichtige Aufgabe des Parsers ist die Erkennung und Behandlung von Syntaxfehlern.

Das folgende Beispiel zeigt die Arbeitsweise eines Parsers:

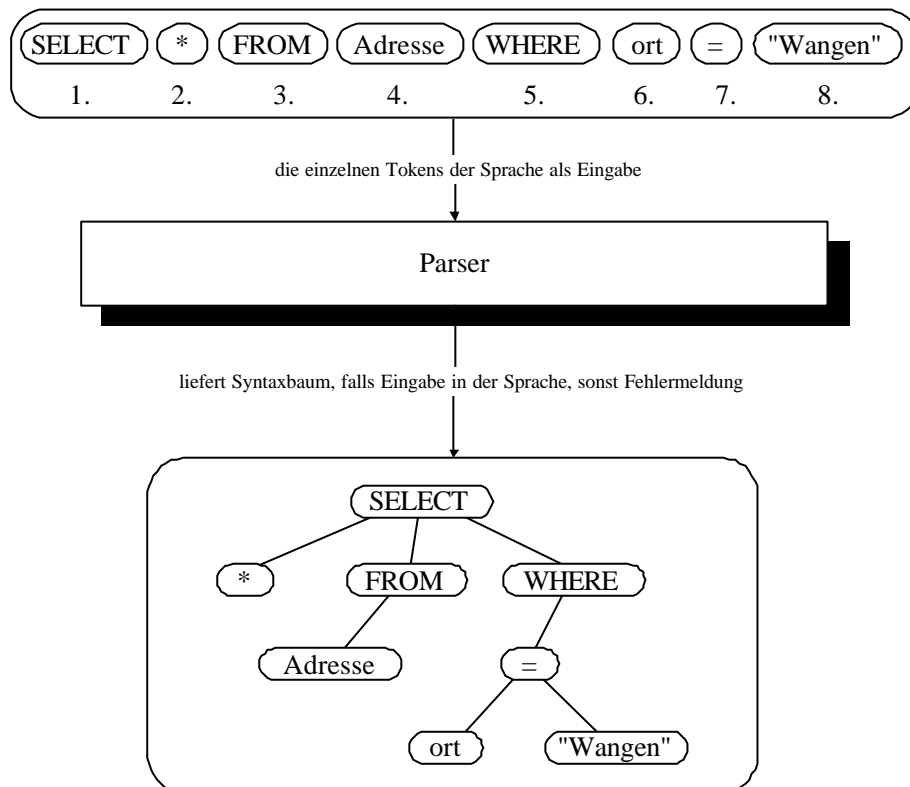


Abbildung 9: Beispiel für die Arbeitsweise eines Parsers

Die syntaktische Struktur aller möglicher Programme einer Programmiersprache läßt sich üblicherweise durch eine kontextfreie Grammatik beschreiben.

### 5.1.1.3. Semantische Analyse

Bestimmte Eigenschaften des Quellprogramms sind nicht durch kontextfreie Grammatiken beschreibbar. Um diese Eigenschaften beschreiben zu können, werden *Kontextbedingungen* verwendet, die im Rahmen der semantischen Analyse überprüft werden. Bei Nicht-Einhaltung der Bedingungen wird üblicherweise mit einer Fehlermeldung reagiert.

Aufgaben der Semantischen Analyse:

1. Die semantische Analyse muß die *Gültigkeitsregeln* für im Quellprogramm deklarierte Bezeichner festlegen, d.h. es muß bestimmt werden, in welchen Teilen des Quellprogramms ein deklariertes Bezeichner existiert.
2. Es müssen die *Sichtbarkeitsregeln* festgelegt werden, d.h. es muß bestimmt werden, wo ein deklariertes Bezeichner in seinem Gültigkeitsbereich sichtbar bzw. verdeckt ist.
3. Es muß die *Typkonsistenz* garantiert werden, d.h. es muß garantiert werden, daß zur Laufzeit Operationen nur auf die für die Operanden definierten Argumenttypen angewandt werden.

### 5.1.2. Synthesephase

Die Synthesephase gliedert sich meistens in die drei folgenden Phasen (vgl. [Aho92]):

1. *Zwischencode-Erzeugung*:  
In vielen Compilern wird nach der Analysephase eine explizite Zwischendarstellung des Quellprogramms erzeugt. Die Zwischendarstellung sollte sich leicht in die Zielsprache übersetzen lassen und leicht zu erzeugen sein.
2. *Code-Optimierung*:  
In dieser Phase wird versucht, den Code der Zwischendarstellung zu verbessern. Das Ergebnis sollte ein effizienterer Code sein.
3. *Code-Erzeugung*:  
In der letzten Phase wird aus der Zwischendarstellung der Zielcode erzeugt.

Nach dieser kurzen Einführung in die Phasen des Compilerbaus wird nun das Werkzeug Java Compiler Compiler (JavaCC) vorgestellt.

## 5.2. Das Werkzeug JavaCC

Das Programm JavaCC (Java Compiler Compiler) der Firma Suntest [JavaCC] ist ein Werkzeug für Java, das aus einer (nahezu vollständig deskriptiven) Grammatikbeschreibung automatisch einen LL(n)-Parser mit Scanner erzeugt.

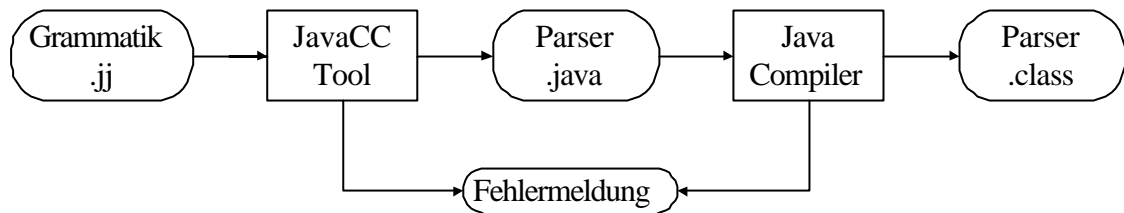


Abbildung 10: Verwendung des Tools JavaCC

Das Tool wird wie folgt verwendet (siehe auch obige Abbildung):

1. Zuerst wird eine Grammatikdatei mit dem Suffix `.jj` erstellt, in der die zu beschreibende Grammatik mit einigen Zusatzinformationen - wie etwa Programmcode - dargestellt ist. Es findet dabei keine Trennung zwischen Parser und Scanner statt.
2. Das Werkzeug JavaCC erstellt mit der Grammatikdatei als Eingabe mehrere Java-Quelldateien in denen der Scanner und der Parser (speziell in der Datei `Parser.java`) für die angegebene Grammatik generiert werden.
3. Mit Hilfe des Java Compilers, wird der generierte Quellcode zu einer (üblicherweise) ausführbaren Klasse kompiliert.

In beiden Übersetzungsphasen können Fehler auftreten: Zum einen kann die Beschreibung der Grammatik fehlerhaft sein, und zum anderen kann Programmcode fehlerhaft sein, der als Zusatzinformation in der Grammatikdatei enthalten ist. Ist der Programmcode fehlerhaft, so wird dies erst von dem Java Compiler entdeckt, da JavaCC keine Überprüfung des Programmcodes auf Korrektheit vornimmt. Dies kann etwas problematisch sein, da die Zuordnung von Fehlern zu deren Position erst bei der späten Erkennung in dem von JavaCC erzeugten Code stattfindet.

Eine JavaCC Grammatikdatei enthält eine optionale Liste von Einstellungen, gefolgt von einer Java Übersetzungseinheit, welche sich zwischen den Schlüsselwörtern `PARSER_BEGIN` und `PARSER_END` befindet. Diese Übersetzungseinheit stellt die Schnittstelle des Parsers zum restlichen System dar. Im Anschluß daran folgt die Definition der einzelnen Tokens und der Produktionsregeln, welche zusammen die Grammatik beschreiben.

### 5.2.1. Ein kleines Beispiel

Die Verwendung des Werkzeugs JavaCC soll an einem kleinen Beispiel demonstriert werden. Es soll ein Parser entwickelt werden, der prüft, ob eine Eingabe bestehend aus öffnenden und schließenden Klammern gültig ist. Gültig sind Eingaben, bei denen die Anzahl der schließenden Klammern gleich der Anzahl der öffnenden Klammern ist, wobei niemals eine schließende Klammer vor der zugehörigen öffnenden erscheinen darf.

Die dafür notwendige JavaCC Grammatikdatei, aus welcher der Parser generiert wird, sieht wie folgt aus:

```
PARSER_BEGIN(Parser)

    public class Parser
    {
        public static void main(String args[])
            throws Exception
        { Parser p = new Parser(System.in);
          p.Teste();
        }
    }

PARSER_END(Parser)

SKIP: { " " | "\t" | "\n" | "\r" }

TOKEN: { <KL: "(">
        | <KR: ")">
        }

void Teste(): { }
{
    (<KL> [Teste()] <KR>)*
}
```

Das Beispiel zeigt die Grundbestandteile einer JavaCC Grammatikdatei:

1. *Java-Übersetzungseinheit:*

Die Java-Übersetzungseinheit definiert die Klasse, in die der Parser generiert werden soll. In der Java-Übersetzungseinheit kann die Schnittstelle zum restlichen System implementiert werden. In diesem Beispiel wird eine Eingabe von der Tastatur gelesen, welche von dem Parser überprüft wird. Als Startsymbol für die Produktionen wird die Produktionsregel `Teste` verwendet, dies wird durch den Aufruf der Methode `Teste` verdeutlicht.

2. *Ignorierbare Zeichen:*

Mit dem Schlüsselwort `SKIP` wird angegeben, daß die Zeichen, die nach diesem Schlüsselwort folgen, von dem erzeugten Scanner ignoriert werden sollen. In obigem Beispiel werden Leer-, Tabulator- und Zeilenvorschubzeichen ignoriert.

### 3. *Definition der Tokens:*

Mit dem Schlüsselwort `TOKEN` beginnt die Definition der einzelnen Tokens der Sprache. Obiges Beispiel definiert als Tokens die öffnende und die schließende Klammer. Dabei wird das Token der öffnenden Klammer mit `KL` und das der schließenden Klammer mit `KR` bezeichnet wird.

### 4. *Angabe der Produktionen:*

In JavaCC wird eine Produktion als eine Art Java-Methode angegeben, wobei Java-Code insbesondere für Aktionen eingeflochten werden kann. Der Name der Methode gibt dabei das auf der linken Seite einer Produktion stehende Nichtterminalsymbol an. Zwischen den ersten geschweiften Klammern können etwaige Parameter übergeben werden. Dies ist nur dann notwendig, wenn Java-Code mit in die Grammatik aufgenommen wird. Zwischen den zweiten geschweiften Klammern kann die rechte Seite der Produktion angegeben werden. Dabei dient der `*` für die Kleenesche Hülle, `+` steht für die positive Hülle und `[ ]` gibt optionale Ausdrücke an. Nichtterminalsymbole müssen wie Methodenaufrufe angegeben werden; um Terminalsymbole müssen spitze Klammern angebracht werden. Innerhalb der zweiten geschweiften Klammern kann auch Java-Quellcode für etwaige Aktionen eingeflochten werden.

Eine genaue Beschreibung, wie eine JavaCC Grammatikdatei syntaktisch aufgebaut sein muß, findet sich in dem Anhang wieder.

Das Werkzeug JavaCC erlaubt es lokale Lookaheads zu verwenden. Dadurch wird sogar das Parsen von kontext-sensitiven Grammatiken möglich. Ein Nachteil der bisherigen Version von JavaCC ist, daß es keine links-rekursiven Produktionen erlaubt. (in der neuen Beta-Version ist dieses Manko behoben!)

Das Ergebnis eines Parse-Vorgangs (Aufruf der für das Startsymbol definierten Methode) ist die Rückgabe eines abstrakten Syntaxbaumes, der alle verwendeten Produktionen als Knoten enthält. Diese Knoten sind entweder vom vordefinierten Typ `SimpleNode` oder jede Produktion bekommt einen eigenen Typ (durch Verwendung des Werkzeugs `JJTree`).

Für die Generierung eines individuellen Syntaxbaumes wird das nun folgende Werkzeug `JJTree` erläutert.

### 5.3. Das Werkzeug JJTree

Das Werkzeug JJTree arbeitet als Präprozessor für JavaCC. Es erstellt aus einer speziellen JJTree Grammatikdatei, die mit *Informationen* angereichert ist wie die Generierung des Syntaxbaumes erfolgen soll, eine JavaCC Grammatikdatei, welche die angegebene Grammatik enthält angereichert mit *Befehlen* (in Form von Java-Code) für die Generierung des Syntaxbaumes. Die erstellte Datei wird mit Hilfe des Werkzeugs JavaCC zu Java-Quellcode übersetzt. Dieser kann mit einem Java-Compiler in interpretierbaren Byte-Code übersetzt werden. Die folgende Abbildung zeigt diese Vorgehensweise:



Abbildung 11: Verwendung des Tools JJTree

Die JJTree-Grammatikdatei, welche das Suffix `.jjt` haben muß, beschreibt die Grammatik, für die ein Parser mit Scanner generiert werden soll, genau in der Art und Weise wie dies für die Grammatikdatei in JavaCC getan wird. Zusätzlich zu der Grammatikbeschreibung können in der JJTree-Grammatikdatei spezielle Informationen für die Generierung des Syntaxbaumes verankert werden:

JJTree kennt zwei unterschiedliche Betriebsmodi, den einfachen und den Mehrbetriebs-Modus. Im einfachen Modus wird jeder Knoten im Syntaxbaum eine Instanz von der konkreten Klasse `SimpleNode`. Im Mehrbetrieb-Modus wird für jeden Knoten im Syntaxbaum eine Instanz einer speziellen Unterklasse von `SimpleNode` verwendet. Dies hat den Vorteil, daß jedes Konstrukt in einer eigenen Klasse repräsentiert wird, was die Implementierung bzw. Übersetzung des Konstruktes übersichtlicher macht.

Üblicherweise generiert JJTree Code, um für jedes Nichtterminalsymbol in der Sprache einen Knoten im Syntaxbaum zu erstellen. Es gibt aber auch die Möglichkeit, explizit anzugeben, wie die Generierung von Knoten für den Syntaxbaum erfolgen soll. Je nachdem, unter welchen Bedingungen ein Knoten in dem Syntaxbaum erstellt werden soll, unterscheidet man in:

1. *Definitive Knoten:*

Ein neuer Knoten kann mit einer genau vorher spezifizierten Anzahl  $n$  von Kindern erstellt werden. Intern läuft dies so ab, daß  $n$  Knoten vom Stapel genommen werden, und zu Kindern des neu generierten Knotens deklariert werden. Der neu generierte

Knoten wird daraufhin auf den Stapel gelegt. Ein Knoten, der auf die obige Art erstellt werden soll, wird syntaktisch wie folgt angegeben:

```
#DefinitiverKnoten(IntegerAusdruck)
```

Es wird ein Knoten mit dem Namen `DefinitiverKnoten` erstellt, der so viele Söhne hat, wie der Integer-Ausdruck angibt. Der Knoten mit dem Namen `DefinitiverKnoten` ist dabei eine Unterklasse von `SimpleNode`.

### 2. Bedingte Knoten:

Ein bedingter Knoten wird genau dann erstellt, wenn die zugehörige Bedingung erfüllt ist. In diesem Fall werden *alle* Knoten, die sich auf dem Stapel befinden heruntergeholt. Sie werden dann als Kinder eines neu erzeugten Knotens verankert, welcher wiederum auf den Stapel gelegt wird. Ein bedingter Knoten wird syntaktisch so angegeben:

```
#BedingterKnoten(BoolescherAusdruck)
```

Für die Angabe von bedingten Knoten existieren folgende Abkürzungen:

```
#BedingterKnoten
```

dies steht für `BedingterKnoten(true)`

```
#BedingterKnoten(>n)
```

diese Bedingung ist erfüllt, wenn sich auf dem Stapel mehr als  $n$  Knoten befinden.

Die Kombination von Produktionsregeln der Grammatik mit Aktionen für die Generierung des Syntaxbaumes hat einen etwas unschönen Effekt, was an dem folgenden Beispiel gezeigt wird:

Gegeben sei eine Grammatik  $G = (V, T, P, S)$ , für die  $V$  die Menge der Variablen und  $T$  die Menge der Terminalsymbole sei.  $P$  bezeichne die Menge der Produktionen, und  $S$  das Startsymbol.

$$\begin{aligned} G &= \{ V, T, P, S \} \\ V &= \{ A \} \\ T &= \{ z, +, - \} \\ P &= \{ A \rightarrow z + z, \\ &\quad A \rightarrow z - z, \\ &\quad A \rightarrow z \} \\ S &= \{ A \} \end{aligned}$$

In diesem Fall könnten die Produktionsregeln auch einfacher angegeben werden:

$$P = \{ A \rightarrow z [ (+ \mid -) z ] \}$$

Diese einfache Form ist aber nicht immer möglich, wenn Informationen für die Generierung des Syntaxbaumes mit in die Grammatikbeschreibung aufgenommen werden.

Beispiel:

```
void A(): { }
  { (<z> #konstante)
    [ (<+> <z> #konstante #plus(2)) |
      (<-> <z> #konstante #minus(2))
    ]
  }
```

Obige JJTree-Produktionsregel kann - wegen der Informationen für die Generierung des Syntaxbaumes - nicht einfacher dargestellt werden. Durch das Anreichern von Informationen für die Generierung des Syntaxbaumes kann sich in manchen Fällen die Grammatik aufblähen.

Obwohl JavaCC ein top-down Parser ist, konstruiert JJTree den Syntaxbaum per bottom-up. Dazu wird ein Stapel verwendet, auf den die Knoten nach ihrer Erzeugung gelegt werden. Wenn die Eltern dazu ermittelt wurden, werden die Knoten vom Stapel geholt, und dafür der Knoten der Eltern abgelegt.

JJTree erlaubt mit einer Reihe von Parameter-Einstellungen, die Generierung des Syntaxbaumes zu beeinflussen. Die Bedeutung einiger Parameter sind in der folgenden Tabelle angegeben:

BUILD_NODE_FILES	Wird dieser Parameter auf den Wert <code>true</code> gesetzt (default), so werden Implementierungen für die Klasse <code>SimpleNode</code> und für alle Knoten, die in der Grammatik verwendet werden erzeugt, allerdings nur, wenn bisher keine Implementierungen dafür existieren.
MULTI	Wird dieser Parameter auf den Wert <code>true</code> gesetzt, so wird für jeden Knoten, der in der Grammatik verwendet wird, eine eigene Unterklasse von <code>SimpleNode</code> bereitgestellt. (Default-Einstellung ist <code>false</code> )
NODE_DEFAULT_VOID	Wird dieser Parameter auf den Wert <code>false</code> gesetzt, so wird für jedes Nicht-Terminalsymbol in der Grammatik ein Knoten im Syntaxbaum erstellt. (Default-Einstellung ist <code>false</code> )
NODE_PACKAGE	Diesem Parameter kann eine Zeichenkette zugeordnet werden, die angibt, in welches Package die Klassen der Knoten des Syntaxbaumes gespeichert werden sollen. Wird nichts angegeben, so wird das Package verwendet, in dem sich der Parser befindet.
STATIC	Wird dieser Parameter auf den Wert <code>true</code> gesetzt, so wird der Parser als statisches Objekt angelegt. Dies hat Vorteile im Laufzeitverhalten, weshalb der Wert <code>true</code> als Default-Einstellung gesetzt ist.

## 6. OQL - Parser

Ziel dieses Kapitels ist es, anzugeben wie mit Hilfe des Werkzeugs JavaCC ein OQL-Parser generiert werden kann. Bereits in Kapitel 2 wurde die informelle OQL-Syntaxbeschreibung der ODMG vorgestellt. Diese kann nicht von einem Werkzeug wie JavaCC verwendet werden, da die Operator-Prioritäten nicht in die Syntaxbeschreibung mit aufgenommen wurden. Damit JavaCC den OQL-Parser generieren kann, müssen die Operator-Prioritäten in die Syntaxbeschreibung mit aufgenommen werden. Außerdem müssen Links-Rekursionen aus der Grammatik entfernt werden, da JavaCC Grammatiken nicht bearbeiten kann welche Links-Rekursionen enthalten.

Im folgenden wird die hier entwickelte Grammatik angegeben, welche die Operator-Prioritäten berücksichtigt und mit der das Werkzeug JavaCC den OQL-Parser generieren kann. Auf eine Angabe der Grammatikdatei für JJTree und JavaCC wird hier verzichtet - es wird auf die beiliegende CD-ROM verwiesen.

### 6.1. Schlüsselwörter

OQL kennt die folgenden Schlüsselwörter, wobei bei den Schlüsselwörtern nicht zwischen Groß- und Kleinschreibung unterschieden wird:

ABS	AND	ANY	ALL
ARRAY	AS	ASC	AVG
BAG	BY	COUNT	DEFINE
DESC	DISTINCT	ELEMENT	EXCEPT
EXISTS	FALSE	FIRST	FLATTEN
FOR	FROM	GROUP	HAVING
IN	INTERSECT	LAST	LIKE
LIST	LISTTOSET	MAX	MIN
MOD	NIL	NOT	OR
ORDER	SET	SELECT	SOME
STRUCT	SUM	TRUE	UNION
UNIQUE	WHERE		

Als spezielle Zeichen werden in OQL verwendet:

+	-	*	/
:	>	>=	<
<=	=	!=	.
	,	->	..
(	)	[	]

Die Schlüsselwörter und die speziellen Zeichen wurden in der JavaCC Grammatikdatei als OQL-Tokens angegeben.

## 6.2. Produktionen

Für die Grammatikdatei `Parser.jjt` wurden die folgenden Produktionen entwickelt, die im Gegensatz zu der Grammatik, welche von der ODMG angegeben wurde (siehe Kapitel 2.2), die Operator-Prioritäten berücksichtigt. Die Grammatikdatei `Parser.jjt` enthält zusätzlich zu den folgenden Produktionen auch noch Informationen für die Generierung des Syntaxbaumes. Manche Produktionen mußten wegen der zusätzlichen Informationen für die Generierung des Syntaxbaumes nochmals leicht verändert werden (siehe dazu auf beiliegender CD-ROM die Datei `Parser.jjt`)

Fettgedruckte Zeichen stellen Terminale dar. Das \*-Zeichen steht für die Kleenesche Hülle, das +-Zeichen steht für die positive Hülle. Die in eckigen Klammern angegebenen Ausdrücke sind optional. `[a-b]` steht für genau ein Zeichen von einschließlich a bis einschließlich b. Als Startsymbol der Grammatik dient das Nichtterminalsymbol `start`.

<code>start</code>	→	<code>query &lt;EOF&gt;</code>
<code>query</code>	→	<code>niedrige_prioritaet</code>   <b>select</b> <code>select_teil</code>
<code>select_teil</code>	→	<b>[distinct]</b> <code>attribute_projektion</code> <code>from_teil</code> <code>[where_teil]</code> <code>[group_teil]</code> <code>[having_teil]</code> <code>[order_teil]</code>
<code>attribute_projektion</code>	→	<b>*</b>   <code>(projektion (, projektion)*)</code>
<code>projektion</code>	→	<code>(bezeichner : niedrige_prioritaet)</code>   <code>(niedrige_prioritaet [<b>as</b> bezeichner])</code>
<code>from_teil</code>	→	<b>from</b> <code>variablen_deklaration</code> <code>(, variablen_deklaration)*</code>
<code>variablen_deklaration</code>	→	<code>niedrige_prioritaet [[<b>as</b>] bezeichner]</code>
<code>where_teil</code>	→	<b>where</b> <code>niedrige_prioritaet</code>
<code>group_teil</code>	→	<b>group by</b> <code>partition_attribute</code>
<code>partition_attribute</code>	→	<code>projektion (, projektion)*</code>
<code>having_teil</code>	→	<b>having</b> <code>niedrige_prioritaet</code>
<code>order_teil</code>	→	<b>order by</b> <code>sortierkriterium</code> <code>(, sortierkriterium)*</code>
<code>sortierkriterium</code>	→	<code>niedrige_prioritaet [<b>asc</b>   <b>desc</b>]</code>

## 6. OQL-PARSER

---

niedrige_prioritaet	→	( klassenname ) disjunktion   disjunktion
disjunktion	→	konjunktion ( <b>or</b> konjunktion)*
konjunktion	→	(vergleich   ( <b>exists</b> bezeichner <b>in</b> vergleich:vergleich)   ( <b>for all</b> bezeichner <b>in</b> vergleich:vergleich) ) ( <b>and</b> vergleich)*
some_all_any	→	<b>some</b>   <b>all</b>   <b>any</b>
vergleich	→	relation ( ( <b>like</b> string)   (= [some_all_any] relation)   (!= [some_all_any] relation) )*
relation	→	summe ( (>   >=   <   <=) [some_all_any] summe )*
summe	→	term ( (+   -   <b>union</b>        <b>except</b> ) term )*
term	→	in_term ( (*   /   <b>mod</b>   <b>intersect</b> ) in_term )*
in_term	→	unaer ( <b>in</b> unaer)*
unaer	→	[-   +   <b>not</b> ] elementar
elementar	→	elem ((.   ->) primitiv)* [ [ ( ( <b>select</b> select_teil ])   disjunktion   ( (: disjunktion ])   ] ) ]
elem	→	( ( ( <b>select</b> select_teil)   (( klassenname) disjunktion)   (disjunktion   (, disjunktion)+   (.. disjunktion)   ] ) ) )   ( <b>struct</b> (   bezeichner : niedrige_prioritaet   (, bezeichner : niedrige_prioritaet)*   )   ) )

		(list( disjunktion .. disjunktion )
		(array   set   bag   list)
		( ( )   ( query ( , query)* ) )
		((abs   first   last   exists   unique
		sum   min   max   avg   listtoaset
		element   distinct   flatten
		) ( query ) )
		count ( ( *   query ) )
		primitiv
		true   false   nil
		Integer   Float   Character   String
primitiv	→	bezeichner
		[ (
		[(bezeichner : query)   query]
		( , ((bezeichner : query)   query) )*
		]
		)
		]
bezeichner	→	<buchstabe> (<buchstabe>   <zahl>)*
buchstabe	→	<b>a-z</b>   <b>A-Z</b>   <b>_</b>
zahl	→	<b>0</b>   <b>1</b>   <b>2</b>   <b>3</b>   <b>4</b>   <b>5</b>   <b>6</b>   <b>7</b>   <b>8</b>   <b>9</b>
Integer	→	([ <b>1-9</b> ] zahl*)   <b>0</b>
Float	→	[ <b>0-9</b> ]+ . [ <b>0-9</b> ]+ [exponent]
		. [ <b>0-9</b> ]+
		[ <b>0-9</b> ]+ [exponent]
exponent	→	[ <b>e</b>   <b>E</b> ] [+   -] [ <b>0-9</b> ]+
klassenname	→	String

Die Produktionen für String und Character wurden nicht angegeben, da sie UNICODE Zeichen enthalten können - welche hier nicht explizit angegeben werden können. Ein String kann beispielsweise aus einer nahezu beliebigen UNICODE Zeichenfolge bestehen. Bestimmte Zeichenfolgen dienen jedoch als ESCAPE-Sequenzen und sind daher nicht zugelassen (für weitere Details siehe beiliegende CD-ROM).

### 6.3. Spezielle JavaCC Einstellungen

Wie bereits erwähnt, wurde der OQL-Parser mit dem Werkzeug JavaCC erzeugt. Dabei wurde das Werkzeug JJTree (ein Präprozessor für JavaCC) im Mehrbetriebs-Modus betrieben. Das heißt, für unterschiedliche Konstrukte in der OQL-Grammatik werden unterschiedliche Unterklassen der konkreten Klasse `SimpleNode` verwendet. Dies hat den Vorteil, das jedes Konzept in der eigens dafür vorgesehenen Klasse implementiert werden kann. Problematisch ist dabei, daß nicht zu viele Klassen erzeugt werden dürfen, da sonst eine gewisse Redundanz auftritt. So wurden beispielsweise die Operatoren `+`, `-`, `*` und `/` alle in der gleichen Klasse implementiert. Auf Grund der Ähnlichkeit der Operatoren wäre eine Aufteilung der Operatoren auf mehrere Klassen nicht sinnvoll.

JavaCC selbst wurde mit einem globalen Lookahead von eins betrieben, d.h. es wird, wenn nicht anderes angegeben, immer nur ein Zeichen bei der Syntaxanalyse vorausgeschaut, was eine effiziente Analyse gewährleistet (LL(1)).

## 7. Umsetzung von OQL-Anfragen

In den bisherigen Kapiteln wurden die Grundlagen erarbeitet. Es wurden die Sprachen OQL und SQL sowie das Persistenzframework POLAR<sup>®</sup> erläutert. Im letzten Kapitel wurde eine OQL-Grammatik angegeben, mit der ein OQL-Parser generiert werden kann, welcher OQL-Anfragen in Form von Zeichenketten in eine Zwischendarstellung als Syntaxbaumes bringen kann.

Nunmehr wird untersucht, inwieweit OQL-Anfragen (repräsentiert durch den Syntaxbaum) in SQL-Anfragen übersetzt werden können und inwiefern eine Umsetzung nach SQL unmöglich ist. Bei Unmöglichkeit wird angegeben, wie diese Anfragen - unter Inkaufnahme von Nachteilen - trotzdem ausgewertet werden können.

### 7.1. Probleme

Die Aufgabe dieser Arbeit ist die Umsetzung von OQL-Anfragen an relationale Datenbanken. Der Zugriff auf relationale Datenbanken kann nur mittels der Sprache SQL erfolgen, weswegen das Ziel der Arbeit als die Umsetzung von OQL-Anfragen in eine oder mehrere SQL Anfragen angesehen werden kann - die als Resultat das Ergebnis der OQL-Anfrage liefern.

Es lohnt sich zu untersuchen, worin die grundlegenden Unterschiede zwischen OQL und SQL bestehen. Denn gerade aus diesen Unterschieden ergeben sich die Problemherde für die Umsetzung von OQL-Anfragen in SQL-Anfragen.

Die folgende Liste soll einen Überblick darüber verschaffen - also, wo die Probleme bei der Übersetzung von OQL-Anfragen in eine oder in mehrere SQL-Anweisungen liegen:

- OQL erlaubt, entsprechend der objektorientierten Ideologie, das Aufrufen von Methoden fachlicher Klassen in einer Anfrage. Demgegenüber ist es in SQL überhaupt nicht möglich, Methoden fachlicher Klassen aufzurufen.
- OQL erlaubt die Navigation entlang von Beziehungen. Die Navigation kann dabei beliebig lang sein. Demgegenüber ist SQL als Anfragesprache für RDBS entwickelt worden. Entsprechend dem relationalen Modell gibt es keine Möglichkeiten für die Navigation (zwischen Objekten). Es ist nur der Zugriff auf die Attribute einer Relation möglich.
- Jede SQL-Anweisung muß eine Select From Where Anweisung enthalten. Ganz primitive Anfragen wie beispielsweise 1+6, die in OQL erlaubt sind, sind dadurch nicht direkt nach SQL übersetzbar.

- In SQL gibt es keine Konstruktoren für Listen, Felder, Mengen, Multimengen und fachliche Objekte. Es können zwar mit Hilfe des `INSERT`-Befehls neue Daten in eine Tabelle eingefügt werden, aber nicht innerhalb einer Anfrage - was in OQL möglich ist.
- In SQL gibt es den All- und den Existenzquantor (mit Bedingungen) nicht als eigenes Konstrukt. Diese beiden Quantoren - die in OQL vorhanden sind - müssen unter Verwendung von mehreren einfachen SQL-Konstrukten nachgebildet werden.
- Das Ergebnis einer `Select From Where` Anweisung von SQL ist unterschiedlich mit dem von OQL:  
SQL liefert immer als Anfrageergebnis eine Relation in erster Normalform. Bei OQL ist das Anfrageergebnis unterschiedlich. So können Listen, Felder, Mengen, Multimengen, Verbunde, einfache Objekte oder Literale sowie beliebige Kombinationen aus diesen Konstrukten als Ergebnis zurückgeliefert werden (OQL-Anfrageergebnis ist in erweiterter  $NF^2$ -Form - Non First Normal Form):

Bsp.: 

```
select p, (select * from p.adresse) as a
from Person p
```

liefert als Ergebnis:

```
bag<struct<p:Person, bag<a:Adresse>>>
```

- Die Möglichkeiten Gruppierungen zu erstellen (mittels `group by`), unterscheiden sich bei beiden Sprachen hinsichtlich der Mächtigkeit: In SQL gibt man dem Group-by Operator eine Liste mit mindestens einem Spaltennamen an. Es werden dann alle Einträge zusammengefaßt (man kann sich dies als zusammenklappen vorstellen), bei denen die angegebenen Spalten die gleichen Werte haben. In OQL kann man das Gruppierungskriterium selber frei bestimmen, wie das folgende Beispiel zeigt.

Bsp.: 

```
select *
from Person p
group by GUF:      p.getAlter < 7
               BGF: 7  <= p.getAlter < 18
               VJ:  18 <= p.getAlter
```

Die OQL-Anfrage gruppiert alle Personen in die Gruppen GUF (geschäftsunfähig), BGF (beschränkt geschäftsfähig) und VJ (volljährig).

Ein Problem, bei dem Group by Operator, ist auch, daß OQL implizit den Bezeichner `partition` definiert, der alle Objekte enthält, die zu der gleichen Gruppe gehören.

- Auch bei den Möglichkeiten, wie ein Anfrage-Ergebnis sortiert werden soll, unterscheiden sich die beiden Sprachen: In SQL kann nur nach Spalten sortiert werden, hingegen läßt sich bei OQL das Sortierkriterium selbst bestimmen:

Bsp.:            `select     *`  
                 `from        Person p`  
                 `order by p.einkommen - p.ausgaben asc`

Diese OQL-Anfrage sortiert alle Personen nach der Differenz aus Einkommen und Ausgaben.

- OQL ist wie bereits in Kapitel 2 erwähnt, größtenteils orthogonal konzipiert, d.h. solange die Regeln des Typsystems der Sprache nicht verletzt werden, können Ausdrücke beliebig miteinander kombiniert werden. Eine Konsequenz ist, daß in OQL-Anfragen Select From Where Anweisungen wieder in jedem Teil einer Select From Where Anweisung geschachtelt werden können. In der hier untersuchten Teilmenge von SQL (siehe Kapitel 3), können Select From Where Anweisungen hingegen nur in dem Where-Teil einer Select From Where Anweisung geschachtelt werden.
- In OQL können beliebig komplexe Ausdrücke umbenannt werden. Demgegenüber ist es in SQL nur möglich, die Relation (Tabelle) und die Attribute (Ergebnisspalten) umzubenennen.

Bsp.:            `select e.b as d`  
                 `from     (select a.ort as b from Adresse a) as e`

Diese OQL-Anfrage benennt das Ergebnis der eingeschachtelten (komplexen) Select From Where Anweisung um. Es wird also nicht nur eine Klasse sondern das Ergebnis eines komplexen Ausdruckes umbenannt.

- In der hier untersuchten Teilmenge von SQL existieren keine Operationen auf Zeichenketten, d.h. Zugriffe auf ein bestimmtes Zeichen oder eine bestimmte Zeichenfolge einer Zeichenkette sind nicht vorhanden. In Kapitel 2 wurde gezeigt, daß es in OQL eine Reihe von Operatoren gibt, die hierfür vorgesehen sind.
- OQL ist als Anfragesprache für objektorientierte Datenbanksysteme konzipiert worden. In objektorientierten Datenbanken gibt es sogenannte Wurzelobjekte, die den Zugang in das Datenbanksystem markieren. In dieser Arbeit soll die OQL-Anfrage an eine relationale Datenbank abgesetzt werden. Bei diesen DBS gibt es keine Wurzelobjekte, vielmehr können auf alle Relationen der DB zugegriffen werden. Es muß also bestimmt werden, wie auf die Objekte, die in dem RDBS gespeichert sind, zugegriffen werden kann. d.h. ein Ersatz für die in OQL notwendigen Wurzelobjekte (sog. *entry points/names*) ist notwendig.

Es folgen einfache Überlegungen darüber, wie - trotz der angegebenen Probleme - die Umsetzung von OQL-Anweisungen realisiert werden kann.

## 7.2. Einfache Überlegungen

In diesem Abschnitt werden diverse Lösungsansätze für die Umsetzung von OQL-Anweisungen untersucht, die als Zeichenketten vorliegen. Aufbauend auf diesen Überlegungen wird später ein grober Entwurf für die Umsetzung von OQL Anweisungen an relationale Datenbanken entwickelt.

### 7.2.1. OQL-Anfrage in eine einzige SQL-Anfrage übersetzen

Ein Lösungsansatz für die Umsetzung von OQL-Anfragen an eine relationale Datenbank besteht darin, die Zeichenkette welche die OQL-Anfrage enthält, mit Hilfe eines Compilers, in eine einzige Zeichenkette mit gleichbedeutender SQL-Anfrage zu übersetzen. Notfalls muß das Ergebnis der SQL-Anfrage nachbearbeitet werden, um es auf die Objekt-Ebene zu bringen, d.h. aus den Ergebnis-Tupeln müssen Objekte erzeugt werden.

Diese Vorgehensweise wird durch die folgende Abbildung verdeutlicht:

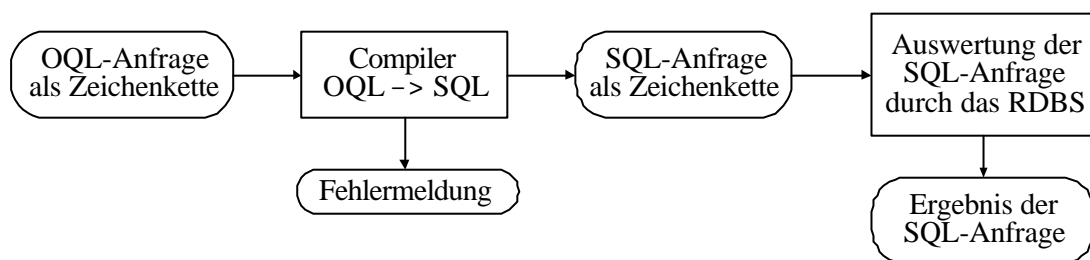


Abbildung 12: Umwandlung der OQL-Anfrage in eine gleichbed. SQL-Anfrage

Mit dieser Vorgehensweise können jedoch nicht beliebige OQL-Anfragen nach SQL übersetzt werden. Im vorherigen Abschnitt wurden Problemfälle angegeben, bei denen eine direkte Umsetzung offensichtlich nicht möglich ist: Es sei hier nochmals erwähnt, daß OQL das Aufrufen von Methoden fachlicher Klassen erlaubt, was in SQL überhaupt nicht möglich ist. Es kann also keine gleichbedeutende SQL-Anfrage für eine OQL-Anfrage mit fachlichen Methodenaufrufen geben.

Beispiel:

```
select *
from Adresse a
where a.getAlter() >=18
```

Diese OQL-Anfrage kann nicht in eine SQL-Anfrage übersetzt werden, da ein Aufruf der Methode `getAlter()` der fachlichen Klasse `Adresse` in SQL nicht möglich ist.

### 7.2.2. OQL-Anfrage in mehrere SQL-Anfragen übersetzen

Ein gängiges algorithmisches Lösungsverfahren in der Informatik ist das *Divide-and-conquer*-Verfahren, das sich in die beiden folgenden Schritte gliedert (aus [DudInf], S. 175):

1. *Divide-Schritt*: Das Problem wird in zwei oder mehrere möglichst gleich große Teilprobleme derselben Art wie das Originalproblem aufgespalten, die unabhängig voneinander gelöst werden.
2. *Conquer-Schritt*: Die Lösungen der Teilprobleme werden zu einer Lösung des Gesamtproblems zusammengefügt.

Es soll untersucht werden, ob es vielleicht möglich ist, mit Hilfe eines Art Divide-and-conquer Verfahrens, komplizierte OQL-Anfragen in mehrere SQL-Anfragen aufzuspalten, die dann ausgeführt und zu einer Lösung der gesamten OQL-Anfrage zusammengesetzt werden. In der Tat gibt es ein SQL-Konstrukt, das sog. View-Konzept, mit dem Teilanfragen erzeugt werden können die dann in anderen Anfragen verwendet werden können. Die folgende Abbildung zeigt die Idee, wie mehrere einfache SQL-Anfragen zu einer komplexen zusammgebaut werden können:

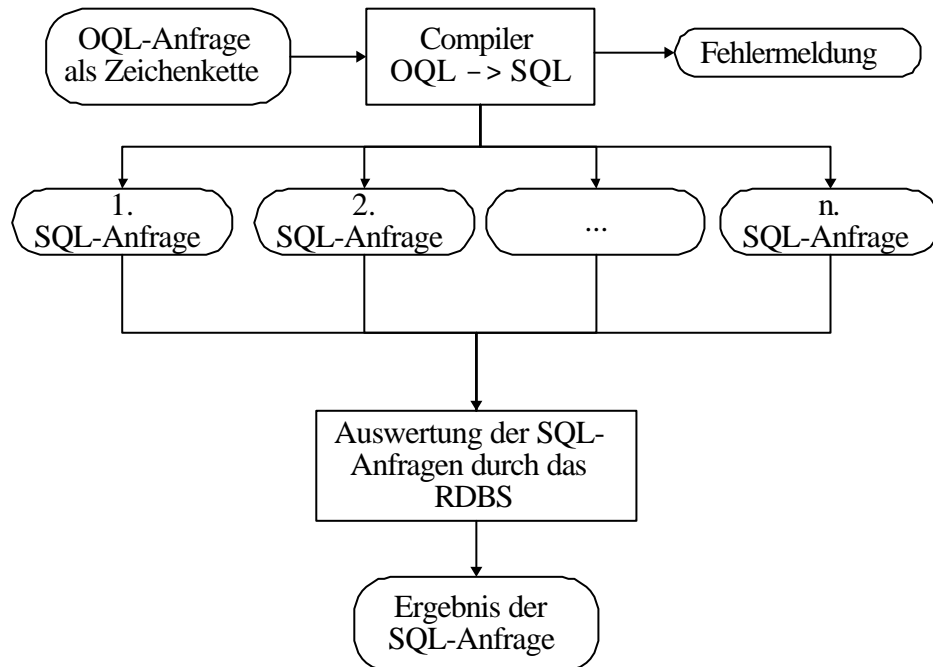


Abbildung 13: Anfrage in mehrere SQL-Anfragen aufsplitten

Der Compiler erzeugt mehrere SQL-Anfragen, wobei die letzte SQL-Anfrage alle anderen SQL-Anfragen verwendet, d.h. die ersten  $n-1$  Anfragen sind sogenannte View-Definitionen, die alle von der  $n$ -ten Anfrage verwendet werden.

Auch hier muß das Ergebnis der SQL-Anfrage nachbearbeitet werden, um es auf die Objekt-Ebene zu bringen.

Man betrachte folgende OQL-Anfrage:

```
select *
from Person, (select * from Adresse) as a
```

Diese Anfrage läßt sich leicht mit Hilfe einer View in die folgenden beiden Anfragen umwandeln:

```
create view a as select * from Adresse

select * from Person, a
```

Der Vorteil, der sich durch das Aufspalten einer OQL-Anfrage in mehrere kleine Teile ergibt, ist relativ bescheiden. Belegt wird dies durch die folgende OQL-Anfrage:

```
select *
from Person as p, (select *
                  from Adresse
                  where Adresse.ort = p.name) as a
```

In der eingeschachtelten Select From Where Anweisung wird die freie Variable `p.name` verwendet. Um diese OQL-Anfrage aufspalten zu können, müßten Views mit Parametern zur Verfügung stehen:

```
create view a(n) as (select * from Adresse
                   where Adresse.ort = n)

select * from Person as p, a(p.name)
```

Views mit Parametern werden jedoch in der hier untersuchten Teilmenge von SQL nicht unterstützt, d.h. obige Anfrage kann nicht erstellt werden. Somit gibt es i.A. Anfragen, die nicht in mehrere Teilanfragen aufgespalten werden können - damit ist dieses Verfahren nur unwesentlich mächtiger als das erste angesprochene Verfahren bei dem die OQL-Anfrage in eine einzige SQL-Anfrage umgesetzt wird.

Es gibt jedoch mindestens einen Spezialfall, bei dem durch das Aufspalten in mehrere Anweisungen eine Umsetzung nach SQL möglich, eine direkte Umsetzung in eine einzige SQL Anweisung jedoch nicht möglich ist. Dieser Spezialfall liegt vor, wenn die eingeschachtelte Select From Where Anweisung das Schlüsselwort `distinct` verwendet und wenn sie unabhängig ist von der umgebenden Select From Where Anweisung, welche das Schlüsselwort `distinct` nicht verwendet. Obwohl mit obigem Lösungsweg - im Gegensatz zur direkten Umwandlung nach SQL - dieser Spezialfall umgesetzt werden kann (s.u.), wurde dieser Lösungsweg wegen der mangelnden Verhältnismäßigkeit von Aufwand und Ertrag

verworfen. Warum dieser Spezialfall bei dieser Methode - und nicht bei der ersten Methode - umgesetzt werden kann, wird weiter unten im detaillierten Entwurf aufgezeigt.

### 7.2.3. OQL-Anfrage interpretieren

Will man Methodenaufrufe von fachlichen Klassen in der OQL-Anfrage verwenden, so kann im allgemeinen die OQL-Anfrage weder in eine einzige noch in mehrere SQL-Anfragen übersetzt werden, da die Methodenaufrufe nicht von dem RDBS ausgeführt werden können.

Man könnte aber kleinere SQL-Anfragen stellen, welche die für die OQL-Anfrage notwendigen Daten des RDBS liefern. Auf diesen Daten könnte dann die OQL-Anfrage ausgewertet (interpretiert) werden. Man würde die Sprache SQL nur noch dazu verwenden, um die benötigten Daten (Objekte) aus der relationalen Datenbank zu holen.

Beispiel:

```
select *
from   Person p
where  p.getAlter() >=17
```

Bei dieser Anfrage könnte die SQL-Anfrage `SELECT * FROM Person` abgesetzt werden, anschließend könnte für jede Person die Where-Bedingung durch Interpretation ausgewertet werden. Als Ergebnis würden die Personen zurückgeliefert werden, welche die Where-Bedingung erfüllen.

Die folgende Abbildung zeigt die Vorgehensweise bei der Interpretation der OQL-Anfrage:

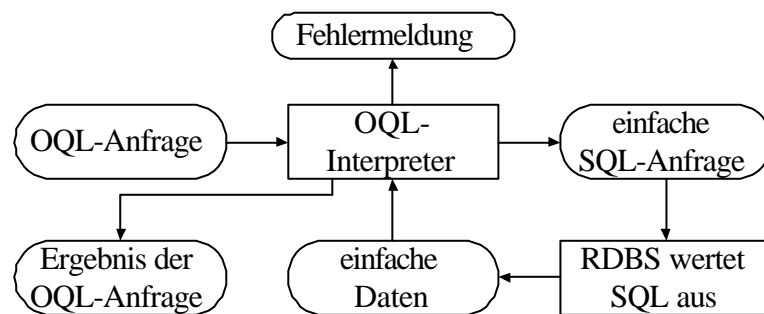


Abbildung 14: OQL-Anfrage interpretieren

Beispiel:

```
select p.getAlter()
from   Person.getPersonen() as p
where  p.getName() like "Duck"
```

Diese OQL-Anfrage verwendet in jedem Teil der Select From Where Anweisung das Konzept der Methodenaufrufe von fachlichen Klassen. Da Methodenaufrufe nicht in SQL abgebildet werden können, kann für obige Anfrage auch kein einziger SQL-Befehl bestimmt werden. Es bleibt nichts anderes übrig, als die Anfrage komplett und unabhängig von SQL zu interpretieren.

Von Interesse wäre auch die Vorgehensweise, daß man Teile der OQL-Anfrage interpretiert, und dann auf das durch die Interpretation ermittelte Ergebnis eine SQL-Anfrage absetzt. Diese Vorgehensweise ist jedoch nicht möglich, da ein außerhalb des RDBS ermitteltes Anfrageergebnis nur in ganz eingeschränkter Form (mit dem `in`-Operator) in die Datenbank gebracht werden kann. Mit dem `in`-Operator können jedoch nur die Daten von Spalten (Mengen) nicht aber von Relationen (Tabellen) in die Anfrage gebracht werden. Durch das Anlegen von Tabellen könnte prinzipiell ein außerhalb des RDBS ermitteltes Anfrageergebnis in das RDBS eingespeist werden. Dies bringt jedoch einen viel zu großen Ressourcenaufwand mit sich, da viele Datenbanksysteme bei der Erstellung von Tabellen auch Indexe für diese Tabellen generieren. Damit können Anfrageergebnisse die außerhalb des RDBS ermittelt wurden nicht mehr in die Datenbank gebracht werden! Somit scheidet eine Vorgehensweise die interpretiert und auf das interpretierte Ergebnis eine SQL-Anfrage absetzt generell aus.

Die bisherigen Überlegungen haben gezeigt: Wenn man Methodenaufrufe von fachlichen Klassen zulassen will, bleibt nichts anderes übrig als die komplette Anfrage zu interpretieren. In manchen Fällen mag es zwar möglich sein, schon eine Reihe restriktiver Bedingungen bei der Datenbeschaffung mit aufzunehmen, im Allgemeinen ist dies jedoch nicht möglich (dieser Punkt wird später noch einmal aufgegriffen).

### 7.2.4. Bewertung

Das erste Verfahren, bei dem die gesamte OQL-Anfrage in eine einzige gleichbedeutende SQL-Anfrage übersetzt wird, ist hinsichtlich des Laufzeitverhaltens das beste Verfahren, denn es werden automatisch die Optimierungsmechanismen des relationalen Datenbanksystems verwendet. Der einzige zusätzliche Aufwand besteht in der Übersetzung der OQL-Anfrage in die SQL-Anfrage. Der Nachteil ist, daß nicht alle OQL-Anfragen nach SQL übersetzt werden können, und daher nicht die volle Funktionalität von OQL zur Verfügung steht.

Das zweite Verfahren, bei dem die OQL-Anfrage in Teil-Anfragen aufgespalten wird, bringt nur in Spezialfällen einen Vorteil zu dem ersten Verfahren, da Select From Where Anweisungen, die freie Variablen enthalten, nicht unabhängig von der gesamten Anfrage ausgewertet werden können. Dies liegt daran, daß in der hier untersuchten Teilmenge von SQL das View-Konzept mit Parametern nicht zur Verfügung steht. Da dieses Verfahren keinen bemerkenswerten Vorteil gegenüber dem ersten Verfahren hat, wurde es verworfen.

Das dritte Verfahren, bei dem die OQL-Anfrage interpretiert wird, ist hinsichtlich des Laufzeitverhaltens das schlechteste. Denn bevor interpretiert werden kann, müssen alle notwendigen Daten durch einfache SQL-Anfragen ermittelt werden, d.h. gleichgültig wie gut der OQL-Interpreter ist, der Aufwand für eine Anfrage ist stets größer als bei einer direkten Anfrage durch SQL an das RDBS.

Das folgende Beispiel verdeutlicht dies:

```
select *
from   Person p
where  p.getAlter()=18
```

Angenommen, in der DB existieren 100.000 Personenobjekte, wobei gerade eine Person achtzehn Jahre alt sein soll. Dann führt die Interpretation der obigen OQL-Anfrage zwangsläufig dazu, daß zuerst einmal alle Personen ausgelesen werden (100.000 Stück!). Auf das Ergebnis wird dann die Methode `getAlter()` angewandt. Als Ergebnis bleibt dann die eine Person die achtzehn Jahre alt ist übrig.

Dieses Beispiel zeigt, daß bei der Interpretation immer zuerst alle notwendigen Daten ausgelesen werden müssen, und diese dann erst ausgesiebt werden können. Der Nachteil dieses Verfahrens ist, daß es bei großen Datenbeständen evtl. nicht möglich sein kann alle notwendigen Daten auszulesen. Dadurch daß zuerst der Datenbestand aufgebaut wird, und dann erst ausgesiebt wird, verschlechtert sich auch das Laufzeitverhalten. Vorteil des Verfahrens ist jedoch, daß es jede beliebige OQL-Anfrage bearbeiten kann. So kann durch die Interpretation auch das Aufrufen von fachlichen Methoden in der Anfrage unterstützt werden.

Wie sich aus dieser Bewertung ergibt, weisen die beiden verbliebenen Verfahren, das erste und das dritte, zwar beide Vorteile aber auch extreme Nachteile auf. Es liegt deswegen nahe, einen Lösungsweg zu konzipieren, der die Vorteile beider Verfahren ausschöpft und dadurch die Umsetzung von OQL-Anfragen in SQL-Anfragen verbessert.

Eine kombinierte Vorgehensweise lautet: Ist es möglich, die OQL-Anfrage in eine einzige SQL-Anfrage zu übersetzen, so wird dies getan, und die erzeugte SQL-Anfrage ausgeführt. In den übrigen Fällen, in denen eine direkte Umsetzung nach SQL nicht möglich ist, wird die OQL-Anfrage unter zur Hilfenahme einfacher SQL-Anfragen interpretiert.

Die kombinierte Vorgehensweise hat den Vorteil, daß der Anwender nicht wissen muß, welche seiner Anfragen direkt nach SQL übersetzt werden können und welche nicht, - es wird ja ganz OQL unterstützt.

Zwar erhöht sich in den Fällen, in denen eine Umsetzung der OQL-Anfrage in eine einzige SQL-Anfrage nicht möglich ist, durch die Interpretation die Ausführungszeit, die zur Bearbeitung der Anfrage notwendig ist - dies ist aber der Preis für die volle OQL-Unterstützung.

### 7.3. Grober Entwurf

Nach den obigen Überlegungen wurde ein grober Entwurf für die Arbeit festgelegt: Die Umsetzung von OQL-Anfragen an relationale Datenbanken erfolgt demnach in drei Phasen:

*1. Phase:*

In der ersten Phase wird die OQL-Anfrage auf lexikalische- und syntaktische Fehler überprüft. Außerdem wird die Anfrage in eine Zwischendarstellung in Form eines Syntaxbaumes gebracht.

*2. Phase:*

In der zweiten Phase wird die OQL-Anfrage auf semantische Fehler untersucht. In dieser Phase wird auch überprüft, ob die OQL-Anfrage in eine einzige, gleichbedeutende SQL-Anfrage übersetzt werden kann oder nicht.

*3. Phase:*

In der dritten Phase wird wenn möglich, die OQL-Anfrage in eine einzige SQL-Anfrage übersetzt. Ergab die Überprüfung der zweiten Phase, daß eine Umsetzung nicht möglich, so wird die OQL-Anfrage interpretiert. Bei der Interpretation werden lediglich einfache SQL-Anfragen an das RBDS geschickt, die dann die notwendigen Daten (Objekte) für die Interpretation zurückliefern.

Die erste Phase wurde bereits im letzten Kapitel behandelt - und mit Hilfe des Werkzeugs JavaCC implementiert. Die zweite und die dritte Phase wird im Rahmen des folgenden Abschnittes behandelt. Eine Zusammenfassung der ersten beiden Phasen, wie dies bei vielen Compilern üblich ist, ist aus folgenden Gründen nicht angebracht:

- In OQL können Bezeichner vor ihrer Definition verwendet werden. Bei einer Kombination der ersten und der zweiten Phase macht dies die semantische Analyse schwieriger. Bei der Verwendung des Bezeichners muß festgelegt werden, von welchen Typen der Bezeichner sein kann. Ist dann aber bei der Definition der definierte Typ des Bezeichners nicht unter diesen möglichen Typen, so wurde der Bezeichner falsch verwendet. Eine Zuordnung des Fehlers zu der Position wird durch die verspätete Erkennung aufwendig.
- Die erste Phase wurde mit Hilfe des Werkzeugs JavaCC implementiert. Würde die erste Phase mit der zweiten Phase zusammengelegt werden, so würde die JavaCC - Grammatikdatei - durch die semantische Analyse - unnötig mit Java-Code aufgebläht und damit undurchsichtig werden.
- OQL-Anfragen sind üblicherweise klein. Die durch die Aufspaltung in zwei Phasen entstandenen Nachteile auf das Laufzeitverhalten können daher vernachlässigt werden.

### 7.4. Detaillierter Entwurf

Dieser Abschnitt gliedert sich in die zwei Hauptpunkte, welche zusammen die oben beschriebene dritte Phase bilden:

1. Übersetzung von OQL-Anfragen in gleichbedeutende SQL-Anfragen
2. Interpretation von OQL-Anfragen

In diesen beiden Hauptpunkten werden auch die für die zweite Phase (semantische Analyse) notwendigen Informationen dargestellt - sofern sie nicht schon in Kapitel 2 dargestellt wurden.

OQL wurde, wie bereits erwähnt, als Anfragesprache für objektorientierte Datenbanksysteme konzipiert. Der Zugriff auf die Daten einer objektorientierten Datenbank erfolgt über sogenannte Einstiegs-Punkte ("*root objects*" bzw. "*entry names*"), die den Ausgangspunkt einer jeden Anfrage bilden. Persistenzframeworks verfügen nicht über solche Einstiegs-Punkte. Als Ersatz dafür wurden in dieser Arbeit die fachlichen Klassennamen verwendet.

#### 7.4.1. Übersetzung von OQL nach SQL

Das primäre Ziel dieser Diplomarbeit ist es, OQL-Anfragen in äquivalente SQL-Anfragen zu übersetzen. Dies ist von Interesse, da durch diese Umwandlung Anfragen auf Objektebene effizient ausgeführt werden können.

Im folgenden wird für die einzelnen Konzepte angegeben wie sie, wenn möglich, nach SQL umgesetzt werden. Außerdem werden die Gründe, warum bestimmte Konzepte nicht nach SQL umgesetzt werden können, explizit angegeben.

##### 7.4.1.1. Select From Where Anweisung

Im Kapitel 2 wurde dargestellt, daß in OQL in jedem Teil einer Select From Where Anweisung eine oder mehrere Select From Where Anweisungen eingeschachtelt werden können, d.h. der Select-Teil kann wiederum eine oder mehrere Select From Where Anweisungen enthalten, gleiches gilt für die Teile From, Where, Group by, Having und theoretisch auch für Order by. Im Gegensatz dazu, erlaubt die hier verwendete Teilmenge von SQL nur, daß im Where-Teil eine oder mehrere Select From Where Anweisungen geschachtelt werden können.

Im folgenden wird gezeigt wie eine Select From Where Anweisung im Syntaxbaum dargestellt wird.

```
select  distinct a1
from    a2
where   a3
group by a4
having  a5
order by a6
```

Diese Anfrage wird wie in folgender Abbildung im Syntaxbaum dargestellt. Wird das Schlüsselwort `distinct` weggelassen, so fehlt einfach der Knoten `distinct`, der ein Sohn des `Select`-Knotens ist. Auch die Teile `where`, `group by`, `having` und `order by` sind optional; werden sie nicht verwendet, so fehlt der entsprechende Knoten mit seinen Söhnen im Syntaxbaum.

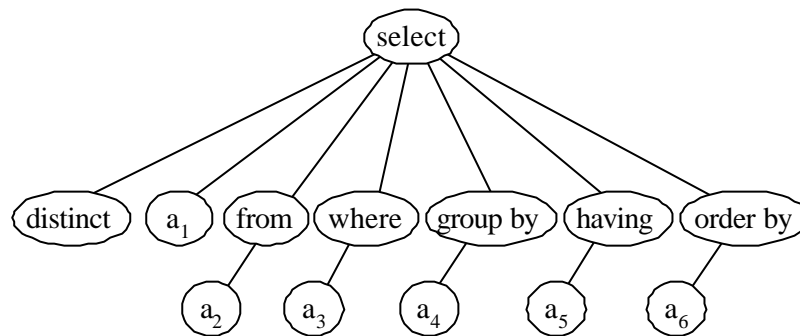


Abbildung 15: Allgemeine Select From Where Anweisung im Syntaxbaum

Bei der Typüberprüfung muß der Typ von  $a_2$  eine Kollektionen sein.  $a_3$  und  $a_5$  müssen beide vom Typ `Boolean` sein.  $a_6$  muß ein oder mehrere Sortierkriterien angeben, wobei diese vom Typ `Comparable` sein müssen.  $a_4$  muß eine oder mehrere Gruppierungs-Attribute angeben. Werden den Gruppen keine expliziten Namen gegeben, so wird ein interner Systemname verwendet. Für jedes Element der Projektion  $a_1$  muß ein Name bestimmbar sein, es dürfen nicht gleiche Namen mehrfach verwendet werden.

Die Teile `Group by`, `Having` und `Order by` werden nicht in diesem Abschnitt behandelt, ihnen wird ein eigener Abschnitt gewidmet.

#### 7.4.1.1.1. Eine einzige Select From Where Anweisung

Wird in der OQL-Anfrage nur eine einzige `Select From Where` Anweisung verwendet, d.h. enthält weder der `Select` noch der `From` oder der `Where`-Teil eine weitere `Select From Where` Anweisung, so kann die `Select From Where` Anweisung direkt nach `SQL` übersetzt werden.

OQL:	→	SQL:
<code>select a<sub>1</sub> from a<sub>2</sub></code>	→	<code>(SELECT a<sub>1</sub> FROM a<sub>2</sub>)</code>
<code>select a<sub>1</sub> from a<sub>2</sub> where a<sub>3</sub></code>	→	<code>(SELECT a<sub>1</sub> FROM a<sub>2</sub> WHERE a<sub>3</sub>)</code>

### 7.4.1.1.2. Select From Where Anweisung im Where-Teil

Wird in einer OQL-Anfrage eine Select From Where Anweisung im Where-Teil einer Select From Where Anweisung verwendet, so kann auch die eingeschachtelte Anweisung wie oben direkt nach SQL übersetzt werden. Die eingeschachtelte Anweisung kann dann im Where-Teil wieder eine Select From Where Anweisung enthalten, usw.

### 7.4.1.1.3. Select From Where Anweisung im From-Teil

Wird in einer OQL-Anfrage eine Select From Where Anweisung in dem From-Teil einer Select From Where Anweisung geschachtelt, so kann die eingeschachtelte Select From Where Anweisungen nicht einfach wie oben nach SQL übersetzt werden. Dies liegt daran, daß wie bereits früher angegeben, die hier untersuchte Teilmenge von SQL das Schachteln von Select From Where Anweisungen im From-Teil einer anderen Select From Where Anweisung nicht erlaubt (MS-Access unterstützt dies nicht).

Was macht eine Select From Where - Anweisung eigentlich im From-Teil einer anderen Select From Where Anweisung? Die eingeschachtelte Anweisung liefert eine Kollektion, die als Ausgangspunkt für die äußere Anweisung dient.

Es stellt sich die Frage, ob dies nicht mit einer einzigen Select From Where - Anweisung möglich ist.

Man betrachte die folgende OQL-Anfrage:

```
select a
  from  (select p.name from Person p) as a
 where a like "Duck"
```

Diese Anfrage kann in folgende gleichbedeutende SQL-Anfrage umgewandelt werden:

```
SELECT p.name
FROM   Person p
WHERE  p.name LIKE "Duck"
```

In bestimmten Fällen kann also eine Select From Where Anweisung, die im From-Teil eine eingeschachtelte Select From Where Anweisungen enthält, in eine einzige Select From Where Anweisung umgewandelt werden.

Das grobe Verfahren, wie die eingeschachtelte Select From Where Anweisung aufgelöst werden kann, wird in der folgender Lösungsidee angegeben.

### *Lösungsidee:*

- Der From-Teil der eingeschachtelten Anfrage wird in den From-Teil der umgebenden Select From Where Anweisung übernommen.
- Der Select-Teil der eingeschachtelten Anfrage wird als Ausdruck für den Aliasnamen (Iteratorvariable) gespeichert, der für die eingeschachtelte Select From Where Anweisung steht.
- Alle Teile der umgebenden Select From Where Anweisung müssen so verändert werden, daß der Aliasname der eingeschachtelten Select From Where Anweisung nicht mehr vorkommt, d.h. der Aliasname für die eingeschachtelte Select From Where Anweisung muß durch den expliziten Ausdruck ersetzt werden.
- Der Where-Teil der eingeschachtelten Anfrage wird in den Where-Teil der umgebenden Select From Where Anweisung übernommen.

Nach dieser kurzen Lösungsidee wird anhand von einem Beispiel gezeigt, wie die im From-Teil eingeschachtelte Select From Where Anweisung aufgelöst werden kann.

### Beispiel:

```
select e.name, a.ort
from   Adresse a,
       (select * from Person p where p.name like "D*") as
e
where  e.vorname like "D*" and (a.ort like "E*")
```

### Vorgehensweise:

1. Schritt: From-Teil der eingeschachtelten Anweisung auflösen  
Der From-Teil der eingeschachtelten Anweisung wird zu dem From-Teil der umgebenden Anweisung hinzugefügt.  
Der neue From-Teil lautet somit: `from Adresse a, Person p`
2. Schritt: Ausdruck für den Aliasnamen festlegen  
Der Ausdruck, der im Select-Teil der eingeschachtelten Select From Where Anweisung steht, wird für den Aliasnamen gespeichert. Im obigen Beispiel steht im eingeschachtelten Select-Teil ein \*, d.h. der Aliasname e steht jetzt für den Bezeichner p, da der \* ja als Synonym für alle Personen (p) steht.
3. Schritt: Aliasnamen der eingesch. Select From Where Anweisung ersetzen  
Überall, wo der Aliasname für die eingeschachtelte Select From Where Anweisung auftritt, muß dieser durch den expliziten Ausdruck ersetzt werden, in diesem Beispiel bedeutet dies, daß e durch p ersetzt werden muß.

### 4. Schritt: Where-Teil der eingeschachtelten Anweisung auflösen

Auch der Where-Teil der eingeschachtelten Anweisung wird zu dem Where-Teil der umgebenden Anweisung hinzugefügt, evtl. mit dem bisherigen Where-Teil durch `and` getrennt.

Es folgt, die komplette erzeugte Anfrage:

```
select p.name, a.ort
from   Adresse a, Person p
where  (p.name like "D*") and (p.vorname like "D*")
       and (a.ort like "E*")
```

Einschränkungen dieser Vorgehensweise: Das Verfahren kann genau dann nicht angewandt werden, wenn im Select-Teil der eingeschachtelten Select From Where Anweisung das Schlüsselwort `distinct` verwendet wird und in der umgebenden Select From Where Anweisung nicht. Der Grund dafür ist, daß in diesem Fall Duplikatzeilen der eingeschachtelten Select From Where Anweisung entfernt werden sollen, was im nachhinein nicht mehr möglich ist.

Beispiel:

```
select *
from   (select distinct a from Adresse a)
```

Dies ist der einzige Fall, in dem mit mehreren SQL Anfragen mehr erreicht werden kann als mit einer einzigen SQL-Anfrage:

Obige Anfrage könnte folgendermaßen in zwei SQL-Anfragen umgesetzt werden:

```
CREATE VIEW x AS (SELECT DISTINCT a FROM Adresse a)
SELECT * FROM x
```

Diese Art die Anfrage aufzuspalten, wurde aber nicht implementiert, da es im allgemeinen nicht möglich, ist die eingeschachtelte Select From Where Anweisung, die `distinct` verwendet, aufzulösen.

Man sieht dies an folgender Anfrage:

```
select *
from   Adresse a,
       (select distinct p
        from   Person p where p.name=a.ort) as x
```

Hier verwendet die eingeschachtelte Select From Where Anweisung die freie Variable a. Da, wie oben bereits erwähnt, keine View-Anfragen mit Parametern zur Verfügung stehen, läßt sich diese Anfrage nicht aufspalten. Es nützt also nicht viel, den obigen Fall zu implementieren.

### 7.4.1.1.4. Select From Where Anweisung im Select-Teil

OQL erlaubt es, daß im Select-Teil weitere Select From Where Anweisungen enthalten sind und damit das Ergebnis nicht mehr in erster Normalform ist. Man betrachte folgende OQL-Anfrage:

```
select a.ort, (select p.vorname
              from   Person p
              where  p.wohnsitz=a.ort) as n
from   Adresse a
```

Diese Anfrage liefert ein Ergebnis in erweiterter NF<sup>2</sup>-Form:

```
bag<struct<ort:string, n:bag<vorname:String>>>
```

Man könnte diese Anfrage nach dem gleichen Prinzip, wie bei der im From-Teil eingeschachtelten Select From Where Anweisung, umsetzen in:

```
SELECT a.ort, p.vorname
FROM   Adresse a, Person p
WHERE  p.wohnsitz=a.ort
```

Dies würde die im Prinzip gleichen Daten, jedoch ohne Hierarchie zurückliefern. Man weiß zwar anhand des Ergebnistyps, welche Hierarchie vorliegen muß, man kann sie jedoch nicht aus den Ergebnis-Tupel wiederherstellen.

Beispiel:

Angenommen obige OQL-Anfrage liefere das folgende Ergebnis:

```
bag<struct<ort:"Entenhausen", n:bag<"Dagobert", "Donald">
      struct<ort:"Entenhausen", n:bag<"Micky">>
>
```

Dann liefert die obige SQL-Anfrage folgende Ergebnis-Tabelle:

a.ort	p.vorname
Entenhausen	Dagobert
Entenhausen	Donald
Entenhausen	Micky

Eine Umwandlung von der Ergebnis-Tabelle in die strukturierte Ergebnismenge ist nicht möglich - denn zu welchem Ort gehört welcher Vorname? Gibt es drei Orte Entenhausen, zwei oder nur einen?

Dieses Beispiel zeigt, daß eine Umwandlung von einer flachen Ergebnismenge in eine Ergebnismenge mit Hierarchieebenen nicht möglich ist - auch wenn genau bekannt ist, wie die Hierarchieebenen aufgebaut sind (s. [Jän]: „Bei Abbildungen von  $R^n \rightarrow R^{n-1}$  geht Information verloren“). Aus diesem Grunde müssen OQL-Anweisungen, die eine Select From Where Anweisung in einem Select-Teil enthalten, interpretiert werden - sie können nicht in eine einzige SQL-Anweisung übersetzt werden, da das Ergebnis wegen der flachen Ergebnismenge (in 1NF) unzureichend ist.

### 7.4.1.2. Attribute und Beziehungen

In diesem Abschnitt wird beschrieben, wie der Zugriff auf Klassen, auf die Attribute einer Klasse sowie auf die Komponenten eines Verbundes umgesetzt wird. Außerdem wird dargestellt, wie die Navigation entlang von Beziehungen nach SQL umgesetzt wird.

#### 7.4.1.2.1. Zugriff auf Klassen, Attribute und Komponenten

Der Zugriff auf eine Klasse kann nach SQL umgesetzt werden, indem statt dem Klassenname der die Klasse repräsentierende Tabellename bzw. Sichtenname verwendet wird. Der Zugriff auf die Attribute einer Klasse kann nach SQL umgesetzt werden, indem statt dem Attributnamen der zugehörige Spaltenname verwendet wird.

In den Beispielen dieser Arbeit wird von einer 1:1 Abbildung von Klassen- auf Tabellename und von Attribut- auf Spaltenname ausgegangen.

Beim Zugriff auf die Komponenten eines Verbundes wird in der generierten SQL-Anfrage - statt dem Verbund - einfach die benötigte Komponente des Verbundes verwendet. Dies ist möglich, da Verbunde nur in der Anfrage selbst erzeugt werden können.

Beispiel:

```
select e.Ort
from   (select Ort: a.ort, Str: a.strasse
        from Adresse a) as e
```

Diese OQL-Anfrage greift auf die Komponente `Ort` des Verbundes `e` (Typ: `struct<Ort:String,Str:String>`) zu. In der zugehörigen SQL-Anfrage wird statt dem Zugriff auf eine Komponente eines Verbundes einfach der entsprechende Wert / Ausdruck verwendet:

```
SELECT a.ort
FROM   Adresse a
```

Hinweis:

In diesem Beispiel wurde die eingeschachtelte Select From Where Anweisung nach dem bereits beschriebenen Konzept aufgelöst.

### 7.4.1.2.2. Navigation zwischen Objekten

Ein wichtiges Konzept von OQL besteht in der Möglichkeit, entlang von Beziehungen von einem Objekt zu anderen Objekten navigieren zu können.

Dieses Konzept existiert in SQL nicht. SQL wurde ja gerade für das relationale Datenmodell entwickelt, deshalb existieren nur Mechanismen für den Zugriff auf die Spalten einer Tabelle (Attribute einer Relation).

In Kapitel 4 wurde erläutert, wie in POLAR<sup>®</sup> Beziehungen auf dem RDBS abgebildet werden. Es wurde angesprochen, daß je nach Beziehungsart eine der beiden Klassen eine Spalte im RDBS für den Fremdschlüssel enthält, der dann auf das entsprechende bzw. die entsprechenden Objekte verweist.

Prinzipiell muß die Navigation zwischen Objekten überall anwendbar sein, egal ob in einem Select-, From-, Where-, Group by-, Having oder Order by- Teil.

Man betrachte folgende OQL-Anfrage, die mittels einfacher Navigation von Objekten der Klasse Adresse zu Objekten der Klasse Person wandert.

```
select a.person.vorname from Adresse a
```

Die Idee für die Umsetzung der Navigation basiert auf einem  $\theta$ -Join mit anschließender Projektion. Die hier erarbeitete Lösung zur Umsetzung der Navigation lautet demnach:

1. Schritt: Die Zielklasse der Navigation wird in den From-Teil übernommen. Dabei wird ein in der Abfrage eindeutiger Aliasname für die Zielklasse verwendet.
2. Schritt: In dem Where-Teil wird die Bedingung aufgenommen, daß Fremdschlüssel gleich Primärschlüssel sein muß (Schritt 1+2 bilden zusammen den  $\theta$ -Join).
3. Schritt: Statt der Objekt-Navigation wird auf die entsprechende Spalte (Attribut) der Zielklasse zugegriffen (Projektion). Wird bei der Navigation nicht auf ein Attribut zugegriffen, so wird auf die gesamte Zielklasse projiziert.

Beispiel:

```
select a.person.vorname  
from Adresse a
```

Diese OQL-Anfrage enthält eine Navigation von der Klasse Adresse zu Person.

Im folgenden wird gezeigt, wie diese Anfrage nach SQL umgesetzt wird:

1. Schritt: Die Zielklasse der Navigation wird in den From-Teil übernommen. Dabei wird ein in der Abfrage eindeutiger Aliasname für diese Klasse verwendet. Der From-Teil der OQL-Anfrage wird also umgewandelt in:

```
FROM Adresse a, Person AS OQLnav1
```

OQLnav1 ist dabei der eindeutige Aliasnamen für die Tabelle Person.

2. Schritt: In dem Where-Teil wird die Bedingung aufgenommen, daß Fremdschlüssel gleich Primärschlüssel sein muß. Es ergibt sich folgender Where-Teil der OQL-Anfrage:

```
WHERE (a.person-id = OQLnav1.oid)
```

3. Schritt: Statt der Objekt-Navigation wird auf die entsprechende Spalte der Zielklasse zugegriffen. Die Objektnavigation `a.person.vorname` wird ersetzt durch `OQLnav1.vorname`.

Es ergibt sich die folgende SQL-Anfrage:

```
SELECT OQLnav1.vorname
FROM   Adresse a, Person AS OQLnav1
WHERE  a.person-id=OQLnav1.oid
```

*Problemfälle:*

- 1) Enthält ein Select-Teil der Select From Where Anweisung den \*- Operator, so muß dieser durch eine explizite Auszählung der einzelnen Spalten ersetzt werden. Dies kommt daher, daß die erzeugte Anweisung mehr Einträge in dem From-Teil hat, als die ursprüngliche, d.h. der Stern (\*) kann nicht beibehalten werden, da wegen dem zusätzlich ausgeführten Join ein falsches Resultat zurückgeliefert werden würde.
- 2) Es ergeben sich aber noch weitere Probleme. Man betrachte dazu folgende OQL-Anfrage:

```
select  a.person.name
from    Adresse a
order  by a.person.name
```

Das obige Lösungskonzept würde folgende SQL-Anfrage generieren:

```
SELECT OQLnav1.name
FROM   Adresse a, Person AS OQLnav1,
```

```
      Person AS OQLnav2
WHERE  (a.person-id=OQLnav1.oid) AND
      (a.person-id=OQLnav2.oid)
ORDER BY OQLnav2.name
```

Diese Anfrage ist aber nicht gleichbedeutend mit oben stehender OQL-Anfrage. Das Problem liegt darin, daß für die gleiche Navigation zwei unterschiedliche Person-Tabellen verwendet wurden, d.h. das zurückgelieferte Ergebnis ist nicht richtig sortiert, da nach dem Tabellennamen OQLnav2 statt nach dem Tabellennamen OQLnav1 sortiert wurde. Damit solche Probleme nicht entstehen, muß in jeder Hierarchie-Ebene für die gleiche Navigation auch der gleiche Tabellennamen verwendet werden.

Unter Beachtung dieser Regel wird folgende SQL-Anfrage generiert, die obiger OQL-Anfrage entspricht:

```
SELECT  OQLnav1.name
FROM    Adresse a, Person AS OQLnav1
WHERE   (a.person-id=OQLnav1.oid)
ORDER BY OQLnav1.name
```

- 3) Ein weiteres Problem ergibt sich bei der Verwendung der Navigation im From-Teil. Man betrachte dazu folgende Anfrage:

```
select a.ort
from   Person p, p.adresse a
```

In diesem Fall darf für die Zielklasse der Navigation kein Aliasname erzeugt werden, vielmehr muß der angegebene Aliasname (in diesem Fall: a) verwendet werden:

```
SELECT a.ort
FROM   Person p, Adresse a
WHERE  (a.person-id=p.oid)
```

Obiges Verfahren läßt sich bei der Navigation über mehrere Objekte genauso anwenden. Man sieht dies an folgendem Beispiel:

```
select p.name
from   Person p
where  p.vater.mutter.name like "D*"
```

Obige OQL-Anfrage enthält eine Navigation über mehr als zwei Objekte. Die Anfrage wird durch mehrmalige Anwendung des obigen Verfahrens umgesetzt in die folgende SQL-Anfrage:

```
SELECT p.name
FROM   Person p, Person OQLnav1, Person OQLnav2
WHERE  (OQLnav2.name like 'D*') AND
```

```
(p.vater-id=OQLnav1.oid) AND  
(OQLnav1.mutter-id=OQLnav2.oid)
```

### 7.4.1.3. Group by und Having

Die Funktionalität der Group by Anweisung von OQL unterscheidet sich, wie bereits dargelegt, grundlegend von der Funktionalität der Group by Anweisung von SQL. Auch das Ergebnis der beiden Anweisungen unterscheidet sich. So liefert die Group by Anweisung von OQL automatisch zu der Gruppierung noch den impliziten Bezeichner `partition` zurück. Dieser Bezeichner enthält für jede Gruppierung die in ihr enthaltenen Elemente. Schon allein durch den impliziten Bezeichner `partition` ist eine allgemeine Umsetzung des Group by Operators von OQL nach SQL nicht möglich. Auch durch die Möglichkeit, daß in OQL die Gruppierungen beliebig definiert werden können, scheidet eine allgemeine Umsetzung aus. Für einige wenige Spezialfälle, ist jedoch eine Umsetzung möglich.

Man betrachte folgende Anfrage:

```
select Ort from Adresse group by Ort: Adresse.ort
```

Diese Anfrage läßt sich nach SQL übersetzen:

```
SELECT Adresse.ort FROM Adresse GROUP BY Adresse.ort
```

Da sich die Umsetzung des Group-by Operators wegen der oben genannten Gründe nur für einige wenige Spezialfälle realisieren läßt, wurde generell auf eine Umsetzung des Group-by Operators von OQL nach SQL verzichtet. Da in OQL der Having-Teil nur in Verbindung mit dem Group-by Teil verwendet werden kann, scheidet daher auch eine Umsetzung des Having-Teils von OQL nach SQL aus. Prinzipiell ließe sich jedoch der Having-Teil genauso wie der Where-Teil direkt nach SQL übersetzen (wenn man von der Verwendung des Bezeichners `partition` absieht).

### 7.4.1.4. Order by

Bereits früher wurde angesprochen, daß in OQL die Möglichkeiten Sortierkriterien zu bestimmen, denen in SQL weit überlegen sind. Dies macht eine allgemeine Umsetzung von OQL- in SQL-Sortierkriterien unmöglich:

```
Beispiel:  select *  
           from Mitarbeiter m  
           order by m.urlaubstage*m.urlaubstage - m.gehalt
```

Das obige, komplexe OQL-Sortierkriterium kann nicht in SQL ausgedrückt werden, da SQL im Order by-Teil als Sortierkriterium nur eine Liste bestehend aus Spaltennamen zuläßt. Enthält das OQL-Sortierkriterium nur einfache Attributzugriffe und Navigationspfade, so kann es dagegen nach SQL übersetzt werden.

Beispiel:

```
select  *
from    Adresse a
order by a.person.name, a.ort desc
```

Diese OQL-Anfrage wird, wie folgt, in SQL umgewandelt:

```
SELECT  a.ort, a.strasse, a.oid, a.postleitzahl
FROM    Adresse a, Person OQLnav1
WHERE   a.person-id=OQLnav1.oid
ORDER BY OQLnav1.name, a.ort DESC
```

Es sei hier nochmals bemerkt, daß Pfad-Navigationen auch in dem Order by Teil einer Select From Where Anweisung nach SQL übersetzt werden können.

### 7.4.1.5. Elementare Ausdrücke

#### 7.4.1.5.1. Unäre Operatoren

Alle von OQL definierten unären Operatoren können direkt von OQL nach SQL übersetzt werden. OQL definiert die folgenden unären Operatoren:

1. Arithmetische unäre Operatoren:  $+$ ,  $-$
2. Bool'sche unäre Operatoren: `not`

Ist  $a$  ein Ausdruck, und  $op \in \{+, -, not\}$  ein unärer Operator, dann wird der Ausdruck  $op\ a$  im Syntaxbaum so dargestellt, daß der Operator  $op = \{+, -, not\}$  als Vater-Knoten den Ausdruck  $a$  als linken Sohn hat:

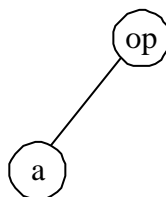


Abbildung 16: Unäre Operatoren im Syntaxbaum

Für die Typüberprüfung und für den durch die Operation erzeugten Typ gelten die folgenden Regeln:

- <INTEGER>      →    <INTEGER>  
 - <FLOAT>         →    <FLOAT>  
 + <INTEGER>       →    <INTEGER>  
 + <FLOAT>         →    <FLOAT>  
 not <BOOLEAN>   →    <BOOLEAN>

Stimmen die Typen überein, so wird der Ausdruck  $op \ a$  wie folgt in SQL überführt:

OQL:	→	SQL:
+ a	→	(+ a)
- a	→	(- a)
not a	→	(NOT a)

### 7.4.1.5.2. Binäre Operatoren

OQL unterstützt die folgenden binären Operatoren:

1. Binäre Operatoren für ganze Zahlen:      +, - , \* , / , mod
2. Binäre Operatoren für Fließkommazahlen: +, - , \* , /
3. Binäre relationale Operatoren:         =, !=, <, <=, >, >=
4. Bool'sche binäre Operatoren:           and, or

Sind  $a_1$  und  $a_2$  zwei Ausdrücke, und  $op \in \{+, -, *, /, \text{mod}, =, !=, <, <=, >, >=, \text{and}, \text{or}\}$  ein binärer Operator, dann wird der Ausdruck  $a_1 \ op \ a_2$  im Syntaxbaum so dargestellt, daß der Operator  $op$  als Vater-Knoten den Ausdruck  $a_1$  als linken- und  $a_2$  als rechten Sohn hat:

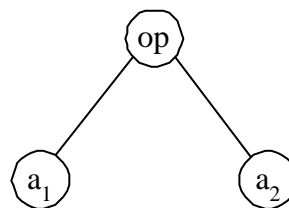


Abbildung 17: Binäre Operatoren im Syntaxbaum

#### 7.4.1.5.2.1. Binäre Operatoren für Ganze- und Fließkommazahlen

Für die Typüberprüfung und für den durch die Operation erzeugten Typ gelten die folgenden Regeln:

Sei  $op \in \{+, -, *, /\}$ :

<INTEGER>	op	<INTEGER>	→	<INTEGER>
<FLOAT>	op	<FLOAT>	→	<FLOAT>
<FLOAT>	op	<INTEGER>	→	<FLOAT>
<INTEGER>	op	<FLOAT>	→	<FLOAT>

Für `op = mod` gilt:

<INTEGER>	op	<INTEGER>	→	<INTEGER>
-----------	----	-----------	---	-----------

Stimmen die Typen überein, so wird der Ausdruck  $a_1 \text{ op } a_2$  wie folgt in SQL überführt:

OQL:	SQL:
$a_1 + a_2$	→ $(a_1 + a_2)$
$a_1 - a_2$	→ $(a_1 - a_2)$
$a_1 * a_2$	→ $(a_1 * a_2)$
$a_1 / a_2$	→ $(a_1 / a_2)$
$a_1 \text{ mod } a_2$	→ $(a_1 \text{ MOD } a_2)$

### 7.4.1.5.2.2. Binäre relationale Operatoren

Damit zwei Operanden  $o_1$  und  $o_2$  mit den Operatoren `=`, `!=`, `<`, `<=`, `>` und `>=` verglichen werden können, müssen diese kompatibel miteinander sein (wurde definiert in Kapitel 2). Bei den Operatoren `<`, `<=`, `>` und `>=` kommt als weitere Anforderung hinzu, daß die Operanden vergleichbar sein müssen, d.h. sie müssen vom Typ `Comparable` sein.

Werden die binären relationalen Operatoren auf die atomaren Literale  $l_1$  und  $l_2$  angewandt, so können diese nahezu direkt nach SQL übersetzt werden.

OQL:	SQL:
$l_1 = l_2$	→ $(l_1 = l_2)$
$l_1 != l_2$	→ $(\text{NOT } (l_1 = l_2))$ bzw. $(l_1 <> l_2)$
$l_1 < l_2$	→ $(l_1 < l_2)$
$l_1 <= l_2$	→ $(l_1 <= l_2)$
$l_1 > l_2$	→ $(l_1 > l_2)$
$l_1 >= l_2$	→ $(l_1 >= l_2)$

Werden die Operatoren `<`, `<=`, `>`, `>=` auf zusammengesetzte Literale (Listen, Felder oder Verbunde) oder Objekte angewandt, so wird eine Fehlermeldung erzeugt, da zusammengesetzte Literale und fachliche Klassen (die von `PObject` abgeleitet sein müssen) nicht vom Typ `Comparable` sind (`PObject` implementiert nicht das Interface `Comparable`). Im Gegensatz zur OQL-Sprachdefinition sind also fachliche Klassen bedingt durch Einschränkungen des Persistenzframeworks POLAR<sup>®</sup> nicht mit den Operatoren `<`, `<=`, `>`, `>=` vergleichbar!

Mengen und Multimengen können mit den Operatoren `<`, `<=`, `>` und `>=` verglichen werden. Dies wird im Rahmen der Mengen-Inklusion weiter unten besprochen.

Die Sprachdefinition von OQL legt nicht fest, wie die Gleichheit von fachlichen Objekten definiert wird. Prinzipiell gibt es zwei Möglichkeiten:

1. Jede fachliche Klasse ist automatisch eine Instanz der Java-Klasse `Object`. Diese Klasse definiert die Methode `equals` die angibt, ob das übergebene Objekt und die aktuelle Instanz der Klasse gleich sind. In diesem Fall kann die Gleichheit von Objekten nur durch Aufruf der Methode `equals` überprüft werden, d.h. eine Umsetzung nach SQL ist nicht möglich. Diese Vorgehensweise wurde bei der Interpretation von OQL gewählt.
2. Die Gleichheit liegt dann vor, wenn die beiden Objekte die gleiche OID haben. Dieser Fall wurde bei der Umsetzung nach SQL implementiert.

Es wurden also beide Möglichkeiten implementiert, um deren Machbarkeit zu zeigen. Damit eine einheitliche Semantik gewährleistet werden kann, muß man sich, bei einer kommerzielle Nutzung, für eine der beiden Möglichkeiten entscheiden.

Die Ungleichheit wird entsprechend gehandhabt. Eine Umsetzung des Vergleichs von Verbunden nach SQL wird nicht unterstützt - sollen Verbunde miteinander verglichen werden, so wird die OQL-Anfrage interpretiert.

### 7.4.1.5.2.3. Bool'sche binäre Operatoren

Für die Typüberprüfung und für den durch die Operation erzeugten Typ gelten die folgenden Regeln:

Sei  $op \in \{\text{and}, \text{or}\}$ :

`<BOOLEAN> and <BOOLEAN> → <BOOLEAN>`  
`<BOOLEAN> or <BOOLEAN> → <BOOLEAN>`

Stimmen die Typen überein, so wird der Ausdruck  $a_1 \text{ op } a_2$  wie folgt in SQL überführt:

OQL:	SQL:
$a_1 \text{ or } a_2$	$\rightarrow (a_1 \text{ OR } a_2)$
$a_1 \text{ and } a_2$	$\rightarrow (a_1 \text{ AND } a_2)$

### 7.4.1.5.3. Operatoren auf Zeichenketten

OQL unterstützt die folgenden Operatoren auf Zeichenketten:

1. Operatoren für die Konkatenation von Zeichenketten:        +, ||
2. Operator der prüft, ob Zeichen c in s vorhanden ist:    c in s
3. Operator zum Zugriff auf ein Zeichen:                    s[i]
4. Operator zum Zugriff auf eine Zeichenkette:            s[i:j]
5. Operator für den Vergleich nach einem Muster:           like

Die binären Operatoren +, ||, like und in werden genauso wie die binären Operatoren im vorherigen Abschnitt im Syntaxbaum repräsentiert. Die Operatoren s[i] und s[i:j] werden, wie folgt, im Syntaxbaum repräsentiert:

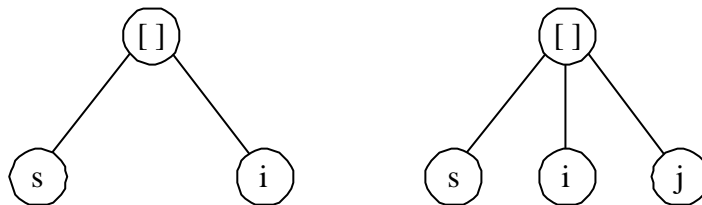


Abbildung 18: Links zeigt den Operator s[i] und rechts den Operator s[i:j]

Für die Typüberprüfung und für den durch die Operation erzeugten Typ gelten die folgenden Regeln:

<STRING> + <STRING>	→	<STRING>
<STRING>    <STRING>	→	<STRING>
<CHARACTER> in <STRING>	→	<BOOLEAN>
<STRING> [ <INTEGER> ]	→	<CHARACTER>
<STRING> [ <INTEGER> : <INTEGER> ]	→	<STRING>
<STRING> like <STRING>	→	<STRING>

Stimmen die Typen überein, so werden die Ausdrücke so nach SQL übersetzt:

OQL:	SQL:
a <sub>1</sub> + a <sub>2</sub>	→ (a <sub>1</sub> + a <sub>2</sub> )
a <sub>1</sub>    a <sub>2</sub>	→ (a <sub>1</sub> + a <sub>2</sub> )
a <sub>1</sub> in a <sub>2</sub>	→ (a <sub>2</sub> LIKE '%a <sub>1</sub> %')
a <sub>1</sub> like a <sub>2</sub>	→ (a <sub>1</sub> LIKE <u>a<sub>2</sub></u> )

In dem String a<sub>2</sub> können bei dem like Ausdruck Wildcards enthalten sein. Da SQL nicht die gleichen Wildcards wie OQL unterstützt, muß die Zeichenkette a<sub>2</sub> für die Verwendung in SQL angepaßt werden, die angepaßte Zeichenkette wird mit a<sub>2</sub> bezeichnet: Die Zeichen \*, ? müssen ersetzt werden. So muß \* durch % und ? durch \_ ersetzt werden. Die hier

untersuchte Teilmenge von SQL erlaubt nicht die Definition von ESCAPE-Zeichen (im Gegensatz zu [ISO], Seite 175). Werden ESCAPE-Zeichen verwendet, so muß die OQL-Anfrage interpretiert werden.

Die Ausdrücke  $s[i]$  und  $s[i:j]$  können nicht nach SQL übersetzt werden. Dies liegt daran, daß die hier untersuchte Teilmenge von SQL keine Möglichkeiten zum Zugriff auf Zeichenketten bietet. Funktionen wie SUBSTRING die in SQL-92 (siehe [ISO], Seite 105) definiert sind, stehen hier nicht zur Verfügung (vgl. Kapitel 3).

### 7.4.1.6. Operationen auf Kollektionen

#### 7.4.1.6.1. Quantoren

Der Allquantor wird syntaktisch durch `for all x in e1:e2`, der Existenzquantor wird durch `exists x in e1:e2` ausgedrückt. Dabei steht  $x$  für einen Variablenname,  $e_1$  für eine Kollektion und  $e_2$  für eine Bedingung (d.h. einen bool'schen Wert). Die folgende Abbildung zeigt die Repräsentation des Existenzquantors und des Allquantors im Syntaxbaum, wobei  $op \in \{\text{for all}, \text{exists}\}$  gilt:

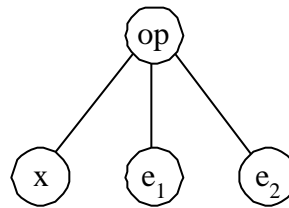


Abbildung 19: Existenzquantor und Allquantor im Syntaxbaum

##### 7.4.1.6.1.1. Der Allquantor

Stößt die Typüberprüfung auf einen Knoten der einen Allquantor repräsentiert, so wird der Syntaxbaum verändert, und zwar so, daß statt dem Knoten des Allquantors ein Teilbaum mit der gleichen Semantik wie der Allquantor angebracht wird. Für die Umwandlung gilt die folgende Regel:

```
for all x in e1:e2 → count(select *
                             from   e1 as x
                             where  not e2) = 0
```

Im Syntaxbaum:

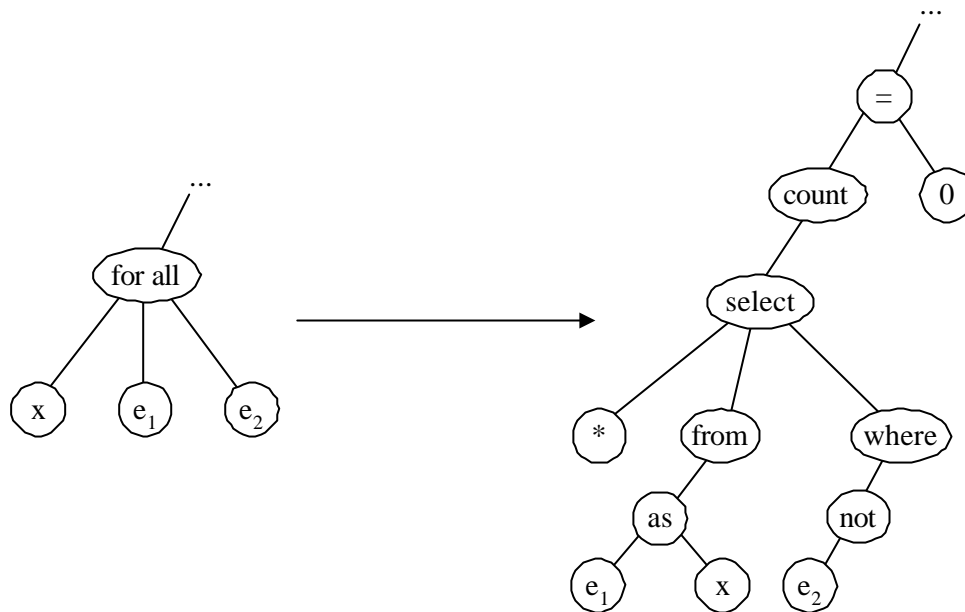


Abbildung 20: Umwandlung des Allquantors

#### 7.4.1.6.1.2. Der Existenzquantor

Stößt die Typüberprüfung auf einen Knoten der einen Existenzquantor repräsentiert, so wird der Syntaxbaum verändert, und zwar so, daß statt dem Knoten des Existenzquantors ein Teilbaum mit der gleichen Semantik wie der Existenzquantor angebracht wird.

```
exists x in e1:e2  →  count(select *
                        from   e1 as x
                        where  e2) > 0
```

Im Syntaxbaum:

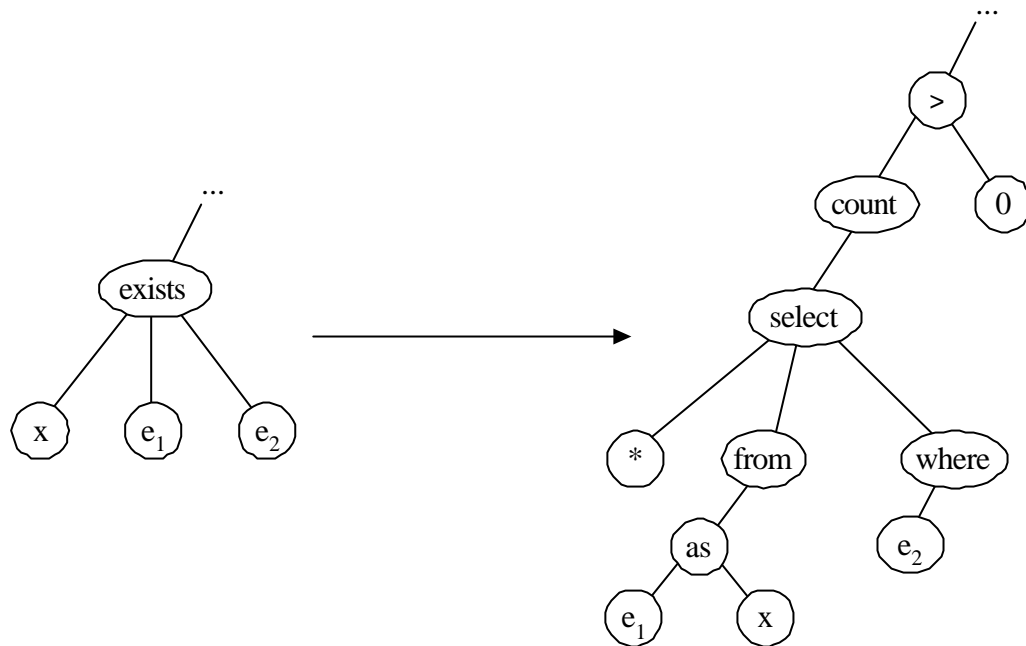


Abbildung 21: Umwandlung des Existenzquantors

#### 7.4.1.6.2. Die Funktionen unique und exists

Für die Typüberprüfung und für den durch die Operation erzeugten Typ gelten die folgenden Regeln:

exists(COLLECTION<T>) → <BOOLEAN>  
 unique(COLLECTION<T>) → <BOOLEAN>

Wird exists auf das Ergebnis einer Select From Where Anweisung x angewandt, so kann die exists Funktion direkt nach SQL übersetzt werden:

OQL:		SQL:
exists(x)	→	EXISTS(x)

Die unique Funktion wird von der hier untersuchten Teilmenge von SQL nicht unterstützt. Sie wird deshalb bereits bei der Typüberprüfung in einen semantisch äquivalenten Ausdruck umgewandelt.

unique(x) → count(select \* from x)=1

Im Syntaxbaum:

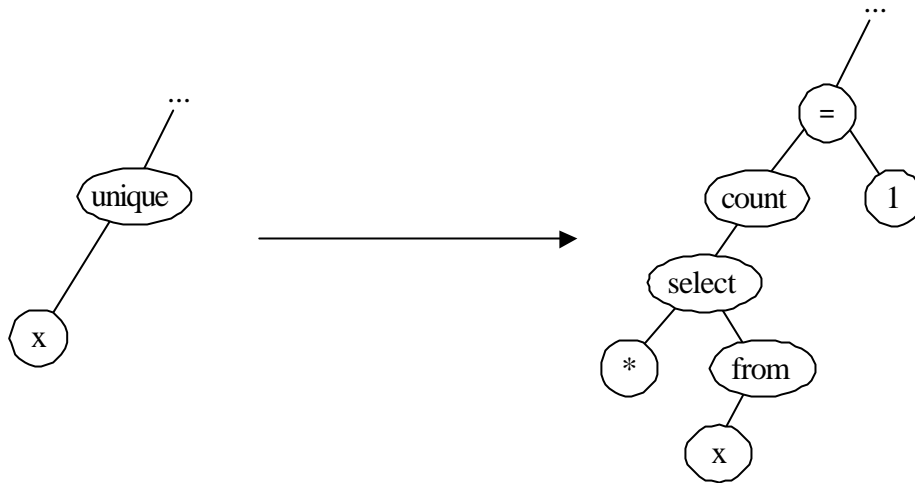


Abbildung 22: Umwandlung des unique-Knotens

#### 7.4.1.6.3. Aggregatfunktionen

Ist  $a$  eine Kollektion, und ist  $f \in \{\text{count}, \text{min}, \text{max}, \text{sum}, \text{avg}\}$  eine Aggregatfunktion, dann wird der Ausdruck  $f(a)$  im Syntaxbaum so dargestellt, daß die Aggregatfunktion als Vater-Knoten den Ausdruck  $a$  als linken Sohn hat:

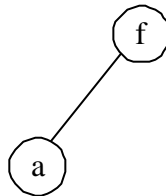


Abbildung 23: Aggregatfunktionen im Syntaxbaum

Für die Typüberprüfung und für den durch die Operation erzeugten Typ gelten die folgenden Regeln:

Sei  $f \in \{\text{min}, \text{max}, \text{sum}, \text{avg}\}$ :

$f(\text{COLLECTION}\langle\text{INT}\rangle)$	$\rightarrow$	$\langle\text{INT}\rangle$
$f(\text{COLLECTION}\langle\text{FLOAT}\rangle)$	$\rightarrow$	$\langle\text{FLOAT}\rangle$
$\text{count}(\text{COLLECTION}\langle\text{T}\rangle)$	$\rightarrow$	$\langle\text{INT}\rangle$ (T beliebig)
$\text{count}(* )$	$\rightarrow$	$\langle\text{INT}\rangle$

Die Aggregatfunktionen können nur dann nach SQL umgewandelt werden, wenn sie auf das Ergebnis einer Select From Where Anweisung angewandt werden. Ist  $a_1$  der Select-Teil der Select From Where Anweisung und steht  $a_2$  für den From-Teil sowie die optionalen Teile Where, Group by, Having und Order by, und ist  $f \in \{\text{min}, \text{max}, \text{sum}, \text{avg}, \text{count}\}$  eine Aggregatfunktion, so kann diese mit folgender Regel nach SQL übersetzt werden:

OQL:	→	SQL:
<code>f(select a<sub>1</sub> from a<sub>2</sub>)</code>	→	<code>SELECT f(a<sub>1</sub>) FROM a<sub>2</sub></code>

Für die Aggregatfunktion `count` gibt es Spezialfälle:

- Entspricht  $a_1$  der Projektion auf eine fachliche Klasse, so ist `select count(a1) from a2` kein gültiger SQL-Ausdruck, da die Aggregatfunktionen von SQL nur auf Spalten nicht aber auf Tabellen (Klassen) anwendbar sind. Statt der Tabelle, die für die Klasse steht, wird die Spalte des eindeutigen Schlüssels (OID) als Argument der SQL - Aggregatfunktion verwendet.
- Entspricht  $a_1$  der Projektion auf einen Verbund, so ist dies auch nicht nach SQL abbildbar. Dies liegt daran, daß die hier unterstützte Menge von SQL nur eine Spalte als Argument der `count` Aggregatfunktion unterstützt.
- Generell muß der Stern-Operator (\*) entfernt werden, da sonst die gleichen Probleme auftreten, die bereits bei der Navigation besprochen wurden.

Beispiele:

OQL:	→	SQL:
<code>count(select a from Adresse a)</code>		<code>(SELECT COUNT(a.OID) FROM Adresse a)</code>
<code>count(select * from Adresse a)</code>		<code>(SELECT COUNT(a.OID) FROM Adresse a)</code>
<code>count(select ort, strasse from Adresse)</code>		<code>(SELECT COUNT(ort) FROM Adresse)</code>
<code>count(select * from Adresse, Person)</code>		<code>(SELECT COUNT(Adresse.OID) FROM Adresse, Person)</code>

### 7.4.1.6.4. Test auf Mitgliedschaft

Der Test auf Mitgliedschaft (`in`-Operator) kann direkt nach SQL übersetzt werden, wenn das linke Argument (und damit die Kollektion des rechten Arguments) kein Objekt einer fachlichen Klasse und kein Verbund darstellt. Liegt ein Objekt oder ein Verbund vor, so muß der `in`-Operator interpretiert werden, d.h. er kann nicht nach SQL übersetzt werden. Dies liegt daran, daß es in SQL keine Verbunde gibt, und daß der `in`-Operator nicht auf Tabellen angewandt werden kann.

OQL:	→	SQL:
$a_1 \text{ in } a_2$	→	$(a_1 \text{ IN } a_2)$

### 7.4.1.6.5. Konkatenation von Kollektionen

Die Konkatenation von Kollektion lässt sich nur nach SQL übersetzen, wenn beide Argumente der Konkatenation Ergebnisse von Select From Where Anweisungen sind. Zudem darf die Konkatenation nicht in einem Teil einer Select From Where Anweisung eingeschachtelt sein. Sind diese Voraussetzungen gegeben, so kann die Konkatenation von Listen und Arrays wie folgt nach SQL umgesetzt werden:

OQL:	→	SQL:
$a_1 + a_2$	→	$(a_1 \text{ UNION ALL } a_2)$

### 7.4.1.7. Zugriff auf die Elemente einer Kollektion

OQL unterstützt den Zugriff auf ein bestimmtes Element einer Kollektion, wenn die Kollektion eine Reihenfolge für ihre Elemente definiert, d.h. ist die Kollektion eine Liste oder ein Feld, so kann auf bestimmte Elemente der Kollektion zugegriffen werden. SQL arbeitet hingegen Mengen orientiert, d.h. egal ob das Resultat mittels Order by sortiert wurde oder nicht, das Ergebnis ist eine Menge oder eine Multimenge (dies ist eigentlich nicht mit dem Begriff der Menge vereinbar, da es keine sortierten Mengen gibt). Da SQL alle Anfrageergebnisse als Mengen oder Multimengen auffasst, gibt es keine Möglichkeit, auf bestimmte Elemente einer Kollektion zuzugreifen, da die Position eines Elements in der Menge nicht definiert ist. Nur MS-Access erlaubt mit dem `first` und dem `last` Konstrukt, den Zugriff auf das erste bzw. letzte Element eines Anfrageergebnisses. Dies ist aber ein konzeptioneller Fehler, denn das erste und das letzte Element eines unsortierten Anfrageergebnisses ist implementierungsabhängig, d.h. interne Details gelangen hier nach außen.

### 7.4.1.8. Operationen auf Mengen

#### 7.4.1.8.1. Vereinigung, Durchschnitt und Differenz

##### 7.4.1.8.1.1. Vereinigung

SQL bietet durch `union` ein Konstrukt für die Mengen-Vereinigung an. Die OQL Vereinigung kann nur dann nach SQL umgesetzt werden, wenn beide Argumente Ergebnisse von Select From Where Anweisungen sind. Zudem darf die Vereinigung nicht in einem Teil einer Select From Where Anweisung eingeschachtelt sein.

OQL:	→	SQL:
Falls $a_1$ und $a_2$ vom Typ <code>set</code> sind: $a_1$ union $a_2$	→	<code>(a<sub>1</sub> UNION a<sub>2</sub>)</code>
sonst: $a_3$ union $a_4$	→	<code>(a<sub>3</sub> UNION ALL a<sub>4</sub>)</code>

### 7.4.1.8.1.2. Durchschnitt

In der hier untersuchten Teilmenge von SQL gibt es kein explizites Sprachkonstrukt für den Mengen-Durchschnitt. In bestimmten Fällen kann der Mengen-Durchschnitt aber durch Verwendung von einfachen Sprachkonstrukten ausgedrückt werden.

Man betrachte folgende Anfrage:

```
(select a.ort from Adresse a)
intersection
(select b.ort from Adresse b)
```

Diese Anfrage ist gleichbedeutend mit der Anfrage:

```
select a.ort from Adresse a
       where a.ort in (select b.ort from Adresse b)
```

Eine Umsetzung nach diesem Konzept wurde nicht vorgenommen. Wird der Durchschnitt aus zwei Mengen benötigt, so muß dies explizit nach obiger Weise durchgeführt werden - andernfalls wird die OQL-Anfrage interpretiert.

### 7.4.1.8.1.3. Differenz

Auch für die Mengen-Differenz gibt es in der hier untersuchten Teilmenge von SQL kein explizites Sprachkonstrukt. In bestimmten Fällen kann die Mengen-Differenz aber durch Verwendung von einfachen Sprachkonstrukten ausgedrückt werden.

Man betrachte folgende Anfrage:

```
(select a.ort from Adresse a)
except
(select b.ort from Adresse b)
```

Diese Anfrage ist gleichbedeutend mit der Anfrage:

```
select a.ort from Adresse a
       where not (a.ort in (select b.ort from Adresse b))
```

Eine Umsetzung nach diesem Konzept wurde nicht vorgenommen. Wird die Differenz aus zwei Mengen benötigt, so muß dies explizit nach obiger Weise durchgeführt werden - andernfalls wird die OQL-Anfrage interpretiert.

### 7.4.1.9. Konvertierungen

#### 7.4.1.9.1. Umwandlung einer Liste in eine Menge

Prinzipiell gibt es in SQL keine Unterscheidung zwischen Listen und Mengen. Wird aber ein Order-by Teil bei der Select From Where Anweisung verwendet, so kann das Ergebnis dieser Anfrage *e* als eine Liste angesehen werden.

Das Ergebnis könnte wie folgt in eine Menge umgewandelt werden:

```
select distinct e
from e
```

Hierbei ist nochmals zu bemerken, daß der Ausdruck *e* nicht das Schlüsselwort *distinct* enthalten darf.

#### 7.4.1.9.2. Umwandlung einer Kollektion in ein Element

In SQL gibt es keine Möglichkeit um auf ein bestimmtes Element einer Menge zuzugreifen. Es gibt keine Konstrukte wie etwa *first* oder *last*. Dementsprechend kann die Funktion *element* von OQL nicht nach SQL übersetzt werden.

#### 7.4.1.9.3. Entfernen von Duplikaten

In SQL können Duplikatzellen nur unter Verwendung des Schlüsselwortes *distinct* im Select-Teil einer Select From Where Anweisung entfernt werden. Eine anderes Konstrukt, das aus einer Ergebnistabelle alle Duplikatzellen entfernt, gibt es nicht. Dementsprechend ist es nicht möglich die OQL-Funktion *distinct* unabhängig von einer Select From Where Anweisung umzusetzen.

#### 7.4.1.9.4. Ebenen von Kollektionen aus Kollektionen

SQL kann keine Mengen über Mengen als Anfrage-Ergebnis zurückliefern, da SQL das Ergebnis einer Anfrage immer in 1. Normalform zurückliefert. Aus diesem Grunde kann es keine Umsetzung der OQL *flatten*-Funktion nach SQL geben.

### 7.4.1.10. Erzeugen von Objekten, Kollektionen und Verbunden

Das Erzeugen von Objekten, Kollektionen und Verbunden läßt sich nicht nach SQL übersetzen, zumal SQL keine Verbunde und keine Objekte kennt. Durch die SQL-Befehle *UPDATE* und *INSERT* können zwar neue persistente, fachliche Objekte verändert bzw. erzeugt werden. Dies ist aber nicht innerhalb einer SQL-Anfrage bzw. innerhalb einer Select From Where Anweisung möglich, so wie dies in OQL vorgesehen ist.

### 7.4.1.11. Methoden- und Funktionsaufrufe

Methoden- und Funktionsaufrufe, sind wie bereits vielfach angesprochen, nicht nach SQL übersetzbar. Werden Methoden- oder Funktionsaufrufe in einer OQL-Anfrage verwendet, so muß die gesamte OQL-Anfrage interpretiert werden.

### 7.4.1.12. Nicht umsetzbare Konzepte (Zusammenfassung)

Es wurde gezeigt, daß sich die folgenden Konzepte nicht in die hier betrachtete Teilmenge von SQL übersetzen lassen. Die folgenden Konzepte müssen daher aus den angegebenen Gründen interpretiert werden:

- *Fachliche Methoden-, Funktions- und Konstruktoren-Aufrufe*,  
da SQL keine Möglichkeiten für das Aufrufen von fachlichen Methoden, Funktionen oder Konstruktoren zur Verfügung stellt. Damit müssen auch die in OQL eingebauten Funktionen `is_defined` und `is_undefined` interpretiert werden.
- *Einfache Anfragen, die keine Select From Where Anweisung verwenden*,  
weil jede gültige SQL-Anfrage eine Select From Where Anweisung enthalten muß.
- *Erzeugen von Objekten, Verbunden, Listen, Feldern, Mengen und Multimengen*,  
da es die entsprechenden Konzepte in SQL nicht gibt - bzw. nicht in der benötigten Form gibt (Mengen).
- *Eingeschachtelte Select From Where Anweisung im Select-Teil*  
(zur Begründung siehe Kapitel 7.4.1.1.4).
- *Eingeschachtelte Select From Where Anweisung im From-Teil*,  
genau dann, wenn die eingeschachtelte Select From Where Anweisung das Schlüsselwort `distinct` verwendet und die umgebende dieses Schlüsselwort nicht verwendet (zur Begründung siehe Kapitel 7.4.1.1.5).
- *Group-by Teil und Having-Teil*  
(zur Begründung siehe Kapitel 7.4.1.3)
- *Order-by Teil*,  
wenn in seinen Sortierkriterien nicht nur Attributzugriffe und Navigationen zwischen Objekten vorkommen (zur Begründung siehe Kapitel 7.4.1.4).
- *Zugriff auf bestimmte Elemente einer Kollektion*  
da SQL mengenorientiert arbeitet und auf Mengen keine bestimmte Reihenfolge der Elemente definiert ist, d.h. die Funktionen `first`, `last`, `element` und der `[ ]`-Operator sind nicht nach SQL übersetzbar.
- *Funktionen, die Kollektionen verwenden*

Die Argumente der Funktionen `exists`, `union`, `in`, `min`, `max`, `sum`, `avg`, `count` und `sum`, die eine Kollektion als Typ erwarten, müssen Ergebnisse von `Select From Where` Anweisungen sein, da Mengen in SQL nur durch die `Select From Where` Anweisung zurückgeliefert werden können.

– *Entfernung von Duplikaten*

Die von einer `Select From Where` Anweisung unabhängige Entfernung von Duplikaten mittels der Funktion `distinct` wäre nur nach SQL umsetzbar, wenn keine geschachtelten `Select From Where` Anweisungen entstehen würden. Dies ist im Allgemeinen aber nicht möglich, da in SQL Duplikatzeilen nur im Rahmen einer `Select From Where` Anweisung durch Verwendung des Schlüsselworts `distinct` entfernt werden können.

– *Umwandlungs-Funktionen,*

wie `listtoset` und `flatten` da es in SQL weder Listen noch geschachtelte Datentypen gibt.

– *Zugriff auf Zeichen einer Zeichenfolge*

da die hier untersuchte Teilmenge von SQL kein Konstrukt für den Zugriff auf bestimmte Zeichen einer Zeichenkette unterstützt (vgl. Kapitel 3).

– *Vergleich von Klassen,*

mittels den Operatoren `>`, `>=`, `<` und `<=` da das Persistenzframework POLAR für fachliche Klassen keine Ordnung definiert.

– *Mengen-Inklusion, Mengen-Durchschnitt und Mengen-Differenz,*

nur in eingeschränkter Weise (siehe Kapitel 7.4.1.8.1.).

### 7.4.2. Interpretation von OQL

Es gibt - wie bereits erwähnt - Fälle, in denen die OQL-Anfrage nicht in eine oder mehrere SQL-Anfragen übersetzt werden kann. In diesen Fällen kann die Auswertung der OQL-Anfrage nicht allein von dem RDBS bewerkstelligt werden. Vielmehr ist ein eigener OQL-Auswerter (Interpreter) notwendig, der eine doppelte Funktion hat: Er fordert die benötigten Daten bzw. Objekte von dem RDBS mit Hilfe von evtl. sehr einfachen SQL-Anfragen an und wertet dann die so erhaltenen Daten entsprechend der OQL-Anfrage aus.

Beispiel:

```
select *
from   Adresse a
where  a.getOrt() like "E*"
```

Das Ergebnis dieser Anfrage kann nicht allein durch das RDBS ermittelt werden (wegen dem fachlichen Methodenaufruf!). Eine Möglichkeit obige Anfrage auszuwerten ist die folgende:

1. Der OQL-Interpreter holt alle Adressen aus dem RDBS. Dies ist nur mittels einer SQL-Anfrage wie bspw. `SELECT * FROM Adresse` möglich.
2. Der Interpreter führt für alle so ermittelten Adressen die Methode `getOrt()` aus.  
Fängt die von dem Methodenaufruf zurückgelieferte Zeichenkette mit dem Buchstaben E an, so wird die zugehörige Adresse in die Ergebnismenge übernommen.
3. Alle Adressen der Ergebnismenge werden zurückgeliefert.

#### 7.4.2.1. Darstellungsformen

Ziel ist die Verarbeitung einer OQL-Anfrage. Eine nicht unübliche Vorbereitung für die Verarbeitung einer OQL-Anfrage wird von [Voss94] wie folgt beschrieben:

*„Wesentlich für eine Vorbereitung einer solchen Anfrage zur Auswertung ist eine Transformation von der nicht-prozeduralen Form in eine prozedurale.“*

So werden beispielsweise in ([Härd], Seite 359ff) geeignete Darstellungsschemata für die Auswertung einer Anfrage angegeben. Alle diese Schemata sind für eine spätere Anfrageoptimierung ausgelegt. Da für diese Diplomarbeit kein Interesse an einer Implementierung bereits existierender Methoden für die Anfrageoptimierung bestand, konnte - zur Reduzierung der Implementierungsarbeit - auf eine geeignetere Zwischendarstellung als den Syntaxbaum verzichtet werden.

In der Literatur gibt es recht wenige Abhandlungen über die Auswertung von OQL-Anfragen. Aber in den existierenden Arbeiten (z.B. [Naka]) wurden für die optimierte Auswertung von OQL-Anfragen geeignetere Darstellungsschemata (wie etwa Plankalküle oder Graphen) verwendet.

Der hier erarbeitete Interpreter wertet eine OQL-Anfrage anhand des Syntaxbaumes aus. Dies hat gegenüber sonstigen Darstellungsformen den Vorteil, daß der schon ohnehin notwendige Syntaxbaum für die direkte Umwandlung von OQL-Anfragen in gleichbedeutende SQL-Anfragen auch für die Interpretation verwendet werden kann. Es entfällt somit eine Transformation in eine (für die Anfrageoptimierung) geeignetere Darstellungsform.

### 7.4.2.2. Verschiedene Datenmodelle bei der Auswertung

OQL ist eine Anfragesprache für objektorientierte Datenbanksysteme. Daher ist es notwendig die Auswertung einer OQL-Anfrage auf der Objekt-Ebene vorzunehmen. Dafür werden intern die im Objekt Modell (siehe Kapitel 2.1) der ODMG definierten Datentypen verwendet, d.h. zur Auswertung der OQL-Anfrage werden die Datentypen fachliches Objekt, Liste, Feld, Menge, Multimenge, Verbund sowie die elementaren Datentypen wie Ganz-, Fließkommazahl, Zeichenkette und Zeichen benutzt. Da die Daten (Objekte) in einem RDBS gespeichert sind - die Auswertung aber auf Objekt-Ebene erfolgen soll - müssen die von einem RDBS zurückgelieferten Daten, bevor sie bei der Interpretation verwendet werden, in einen entsprechenden Datentyp des ODMG Objekt Modells umgewandelt werden. Die folgende Abbildung zeigt, daß für die Interpretation einer OQL-Anfrage sowohl Daten, die von der Programmiersprache (Ergebnisse von Methodenaufrufen) als auch die von dem RDBS (persistente Objekte) her kommen, umgewandelt werden müßten:

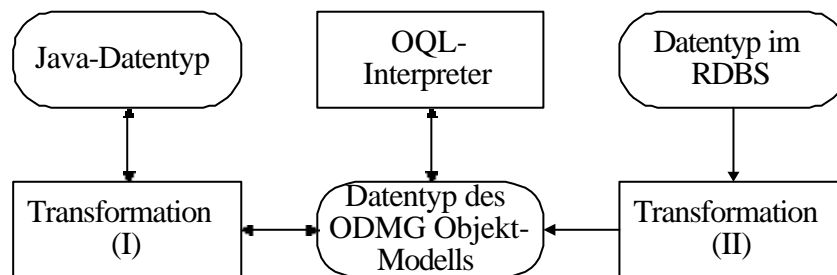


Abbildung 24: Etwaige Umwandlung von Daten bei der Interpretation

Obige Abbildung zeigt beispielsweise, daß Tupel, die von einem RDBS zurückgeliefert werden und die Objekte repräsentieren, für die Auswertung der OQL-Anfrage zuerst in Objekte transferiert werden müssen (Transformation II).

Laut der Zielsetzung der ODMG soll die Sprache OQL die Datentypen, der mit ihr gekoppelten Programmiersprache - in dieser Arbeit Java - verwenden. Der Vorteil dieses Konzepts ist, daß zur Auswertung der OQL-Anfrage die gleichen Datentypen wie in der benutzten Programmiersprache verwendet werden. Dieses abstrakte Ziel, daß OQL nur die von der benutzten Programmiersprache zur Verfügung gestellten Datentypen benutzen soll,

scheitert konkret - bei der Verwendung von Java - daran, daß Java keine Möglichkeiten für die Bildung von Verbunden zur Laufzeit unterstützt. Das Konzept der Verbunde wird aber von OQL benötigt. Klassen können dabei nicht als Ersatz für Verbunde angesehen werden, da neue Klassen nicht zur Laufzeit erzeugt werden können (die Erzeugung von Klassen *on the fly* ist ohne immensen Aufwand nicht möglich). Da es OQL-Anfragen gibt, die als Ergebnis einen Verbund liefern, können nicht nur die eingebauten Datentypen von Java verwendet werden. Vielmehr muß eine Klasse, die Verbunde repräsentiert entwickelt werden.

Probleme gibt es auch bei den Kollektionen: In OQL müssen alle Elemente einer Kollektion den gleichen bzw. einen zueinander kompatiblen Typ aufweisen. In Java existiert diese Restriktion nicht - Kollektionen können über beliebige Element-Typen erzeugt werden.

Wird eine fachliche Methode in einer OQL-Anfrage verwendet, die eine Kollektion zurückliefert, so werden die Elemente dieser Kollektion als Elemente von dem Typ `Object` angesehen. Sollen sie in der OQL-Anfrage weiter verwendet werden, so muß der Typ `Object` mit Hilfe des Cast-Operators in einen spezielleren Typ umgewandelt werden.

Mit Ausnahme dieser zwei Sonderfälle ist eine Umwandlung von Java- in ODMG-Datentypen und umgekehrt nicht notwendig (Transformation I).

### 7.4.2.3. Allgemeine Vorgehensweise bei der Interpretation

Wie bereits erwähnt, wird die Interpretation einer OQL-Anfrage an dem Syntaxbaum vollzogen. Dies ist bei dem `Select From Where` Konstrukt nicht ganz so einfach, was an der nicht-prozeduralen Form dieses Konstruktes liegt. Deshalb wird dieses Konstrukt weiter unten gesondert behandelt.

Bei den anderen Konstrukten besteht die Umsetzung der Interpretation darin, daß die Söhne des jeweiligen Knotens ausgewertet werden, und dann das, zu dem Knoten zugehörige, Konstrukt mit den Söhnen als Argument ausgeführt wird. Als Ergebnis des Knotens wird dann das ausgewertete Resultat zurückgeliefert. Für die interne Auswertung der Anfrage werden - wie oben bereits beschreiben - die ODMG Datentypen verwendet (siehe Kapitel 2.1 bzw. [Catt98]).

Die Konstrukte, die im Syntaxbaum - bei der Umwandlung nach SQL - durch andere Konstrukte ersetzt wurden, werden auch bei der Interpretation von OQL durch die entsprechenden Konstrukte ersetzt. Der Vorteil dieser Vorgehensweise ist, daß für weniger Konstrukte eine Interpretation implementiert werden muß.

Für die Interpretation werden Daten (Objekte) aus dem RDBS benötigt. Bei der Interpretation werden an den Stellen, an denen in der OQL-Anfrage Wurzelobjekte (sog. *entry points*) verwendet wurden, einfache SQL-Anfragen gestellt, die dann die notwendigen Daten aus dem RDBS holen. Diese Daten müssen - ebenfalls wie oben angegeben - zuerst auf die Objekt-Ebene gebracht werden, d.h. aus den einzelnen Spalten die ein fachliches Objekt beschreiben, muß das Objekt selbst erzeugt werden.

Beispiel für die Interpretation einer einfachen OQL-Anfrage:

Es soll der Ausdruck `exists(Adresse)` interpretiert werden. Im Syntaxbaum wird dieser Ausdruck folgendermaßen repräsentiert:

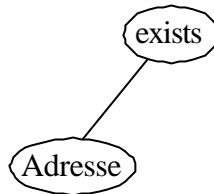


Abbildung 25: Beispiel einer einfachen Interpretation

Die Interpretation beginnt mit dem Wurzelknoten. Für die Auswertung dieses Knotens muß zuerst dessen Sohn ausgewertet werden. Der Sohn des Wurzelknotens ist ein OQL-Wurzelobjekt (*entry point*) der Datenbank, da - wie bereits angegeben - für diese Umsetzung Klassennamen als Einstiegspunkte in die Datenbank angesehen werden. Die Einstiegspunkte können ausgewertet werden, indem eine einfache SQL-Anfrage an das RDBS gestellt wird, die alle Elemente der Relation die für die Klasse steht aus dem RDBS liefert.

In diesem Falle kann die einfache SQL-Anfrage lauten: `SELECT * FROM Adresse`. Das Ergebnis muß in einen Datentyp des ODMG Objekt Modells umgewandelt werden, d.h. die Daten der zurückgelieferten Spalten müssen in Objekte transformiert werden, und diese müssen in eine Kollektion vom Typ `DBag` (s.u.) gesteckt werden. Diese Kollektion wird als Ergebnis des Einstiegspunktes `Adresse` zurückgeliefert. Nun kann der Wurzelknoten im Syntaxbaum ausgewertet werden. Ist der zurückgelieferte `DBag` leer, so wird als Ergebnis das bool'sche Literal `false`, andernfalls wird das bool'sche Literal `true` zurückgeliefert.

### 7.4.2.3. Select From Where

Generell kann die `Select From Where` Anweisung in folgender Reihenfolge ausgewertet werden:

1. From-Teil auswerten
2. Where-Teil auswerten
3. Order by-Teil auswerten
4. Group by-Teil auswerten
5. Having-Teil auswerten
6. Select-Teil auswerten

Dies ist jedoch nicht ganz so einfach, wie das folgende Beispiel zeigt:

```
select *
from Person p, (select *
                 from Adresse a where a.ort = p.name) as q
```

In diesem Beispiel - das eigentlich nicht interpretiert werden müßte - hängt die zweite Kollektion des From-Teils von der aktuellen Iteratorvariable der ersten Kollektion ab, d.h. die zweite Kollektion muß in Abhängigkeit von jedem Element der ersten Kollektion ausgewertet werden. Aus diesem Grunde ist eine Auswertung des From-Teils in einem einzigen Baum-Durchlauf nicht möglich.

Dieses Problem, der Abhängigkeit von Kollektionen im From-Teil, wird durch folgendes Auswertungsverfahren gelöst:

Die Auswertung des From-Teils soll immer nur ein Element der Ergebnis-Kollektion liefern. Es wird also nicht gleich das ganze kartesische Produkt aus den einzelnen Kollektionen im Speicher gebildet, sondern jeweils nur ein Element davon. Will man die komplette Kollektion des From-Teils ermitteln, so muß die Auswertung des From-Teils sooft ausgeführt werden, bis die Auswertung dieses Teiles ein sog. Dummy-Element zurückliefert. Dieses gibt an, daß keine weiteren Elemente mehr für die Kollektion vorhanden sind. Die Ergebnis-Kollektion des From-Teils besteht dann aus allen bisher zurückgelieferten Elementen (mit Ausnahme des Dummy-Elements). Es ist von Vorteil, daß der From-Teil immer nur ein Element zurückliefert, denn dadurch kann für dieses Element sofort die Selektionsbedingung ausgewertet werden, die gegebenenfalls das Element verwirft. Im Speicher wird also nicht das vollständige kartesische Produkt aller Kollektionen gebildet und dann selektiert, sondern es wird jedes Element des kartesischen Produktes einzeln bestimmt und gleich auf die Selektionsbedingung hin überprüft. Elemente, welche die Selektionsbedingung nicht erfüllen, können aus dem Speicher entfernt werden.

Wie kommen die notwendigen Daten in den Interpreter? An den Stellen, an denen in der OQL-Anfrage ein Einstiegspunkt für den Zutritt in die Datenbank verwendet wurde (hier: Klassenname) erzeugt der Interpreter einmalig eine einfache SQL-Anfrage, die alle Elemente dieser Relation (welche die Klasse repräsentiert) zurückliefert. Bei der ersten Auswertung liefert der Knoten das erste Element der Relation zurück (die Reihenfolge der Elemente in der Relation ist dabei beliebig), bei der zweiten Auswertung das zweite, usw. Werden mehrere Einstiegspunkte verwendet, so werden die Elemente - wie bei dem Backtracking-Verfahren - zurückgeliefert, d.h. jedes Element der einen Kollektion wird mit jedem Element der anderen Kollektion kombiniert. Um den kompletten From-Teil auszuwerten muß daher der From-Teil sooft ausgewertet werden, wie die Potenzmenge aus den einzelnen Kollektionen des From-Teils Elemente hat.

Obige Anfrage wird in dem folgenden Syntaxbaum repräsentiert, dabei sind die Einstiegspunkte durch eine Verbindung zu dem RDBS gekennzeichnet:

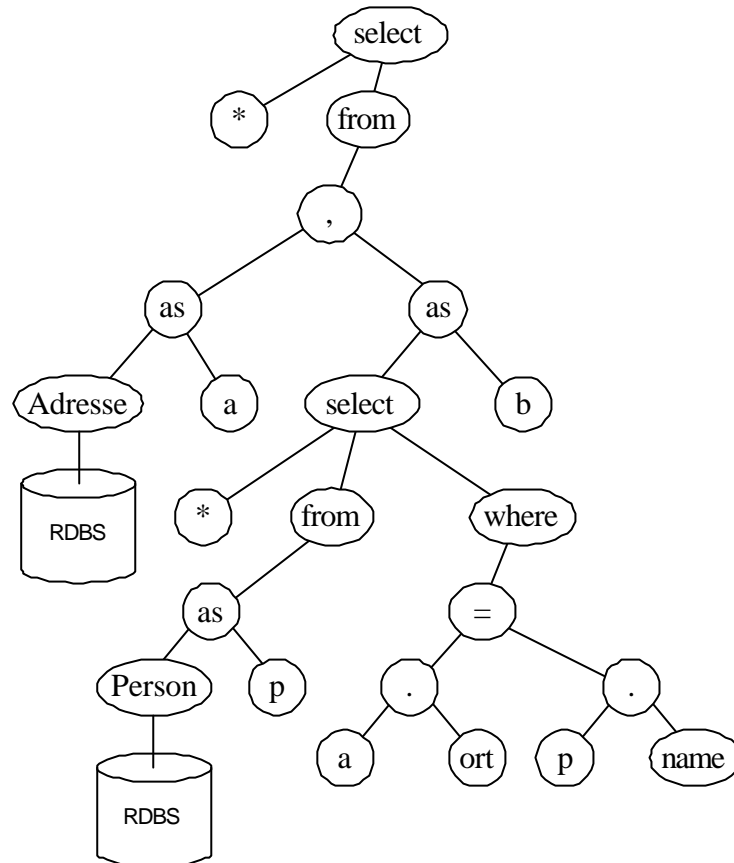


Abbildung 26: Beispiel für die Interpretation einer Anfrage am Syntaxbaum

Die Interpretation des From-Teils dieser Anfrage läuft folgendermaßen ab:

- Der From-Teil besteht aus zwei Kollektionen. Es muß zuerst die linke Kollektion, d.h. der linke Teilbaum unter dem Komma-Knoten ausgewertet werden.
- Bei der Auswertung des linken Teilbaums unter dem Komma-Knoten stößt man auf den Einstiegspunkt Adresse. Für die Interpretation dieses Knotens muß einmalig folgende Anfrage an das RDBS erfolgen: `SELECT * FROM Adresse`. Der Einstiegspunkt Adresse liefert jetzt aber nicht die ganze Relation, sondern nur das erste Element (Tupel) dieser Relation zurück.
- Für dieses Element wird jetzt der rechte Teilbaum unter dem Komma-Knoten ausgewertet. Dabei stößt man auf den Einstiegspunkt Person. Für die Interpretation dieses Knotens muß ebenfalls einmalig eine Anfrage an das RDBS erfolgen, die alle Objekte der Relation Person zurückliefert. Als Ergebnis des Einstiegspunktes Person wird jetzt - wie oben - nur das erste Element der Relation Person zurückgeliefert.
- Bisher wurde der linke und der rechte Teilbaum des Komma-Knotens ausgewertet. Als Ergebnis des Komma-Knotens wird jetzt ein Verbund, bestehend aus dem Ergebnis des linken und des rechten Teilbaumes, gebildet. Das Ergebnis ist somit das erste Element der Kollektion, welches von dem From-Teil zurückgeliefert wird.
- Um alle Elemente der Kollektion des From-Teils ermitteln zu können, muß der From-Teil mehrmals ausgewertet werden. Dazu muß vorerst nur der rechte Sohn des

Komma-Knotens neu ausgewertet werden, wobei der Einstiegspunkt Person jetzt das nächste Element der Relation Person zurückliefert. Das Ergebnis des linken Sohnes des Komma-Knotens bleibt erhalten.

- Solange noch nicht alle Elemente des Einstiegspunktes Person zurückgeliefert wurden, geht die erneute Auswertung des From-Teils - wie eben beschreiben - vor sich. Dabei wird immer das nächste Element des Einstiegspunktes Person mit dem aktuellen Element des Einstiegspunktes Adresse zurückgeliefert. Enthält der Einstiegspunkt Person keine Elemente mehr, so wird ein Dummy-Element zurückgeliefert. Dieses Element signalisiert dem Komma-Knoten, daß nun der linke Teilbaum neu ausgewertet werden muß, was eine neue Auswertung des rechten Teilbaumes nach sich zieht, wobei dann wieder das erste Element des Einstiegspunktes Person zurückgeliefert wird (vgl. Backtracking-Verfahren).
- Enthält auch der Einstiegspunkt Adresse keine Elemente mehr, so liefert auch dieser ein Dummy-Element zurück. Dies signalisiert dem Komma-Knoten, daß jetzt alle Elemente des kartesischen Produktes aus den einzelnen Kollektionen erzeugt wurden. Das Ergebnis ist ein Dummy-Knoten, welcher der Select From Where Anweisung signalisiert, daß bereits alle Elemente des From-Teils zurückgeliefert wurden.

Bei der Auswertung der Select From Where Anweisung wird sofort für jedes Element das von dem From-Teil zurückgeliefert wird, die Bedingung im Where-Teil ausgewertet. Ist sie nicht erfüllt, kann das Element verworfen werden. Wurden alle Elemente des From-Teils bestimmt, so liegt eine Kollektion aus Elementen vor, die alle den Where-Teil erfüllt haben. Wurde ein Order-by Teil angegeben, so kann diese Kollektion entsprechend dem angegebenen Sortierkriterium sortiert werden - das Ergebnis ist eine Liste (und somit wieder eine Kollektion). Wurde eine Gruppierung angegeben, so kann die Kollektion entsprechend den angegebenen Gruppierung in Gruppen aufgespalten werden - das Ergebnis ist wiederum eine Kollektion. Für diese Kollektion wird jetzt noch - falls vorhanden - der Having-Teil überprüft. Den Schluß der Auswertung der Select From Where Anweisung bildet die Auswertung des Select-Teils, der jedes Element der Kollektion auf bestimmte Werte projiziert.

### 7.4.2.4. Methodenaufrufe

Methodenaufrufe werden mit Hilfe der Reflection Möglichkeiten von Java implementiert. Bei der Typüberprüfung wird getestet, ob ein Methodenaufruf vorliegt, indem via Reflection nachgeschaut wird, ob eine Methode mit dem angegebenen Namen existiert. Gibt es eine solche Methode, so werden - wenn vorhanden - die Parameter überprüft, d.h. es wird geprüft, ob die Typen der Parameter der OQL-Anfrage mit den tatsächlich benötigten Parametern für die Methode übereinstimmen. Stimmen die Parameter überein, so kann bei der Auswertung der Anfrage die Methode durch Reflection aufgerufen werden. Zu beachten ist dabei, daß vor dem Methodenaufruf die Parameter der Methode, die ja in einer internen Form vorliegen, in Java-Datentypen umgewandelt werden müssen. Dementsprechend muß das Ergebnis eines Methodenaufwurfes wieder in eine für die Interpretation notwendige interne Darstellung umgewandelt werden. Die umgewandelten Ergebnisse eines Methodenaufwurfes

werden dann bei der Auswertung der Anfrage weiter verwendet. Liefert eine Methode keinen Funktionswert, so wird als Ergebnis der Methode das Literal `nil` verwendet.

Liefert eine Methode ein Feld zurück, so ist nicht klar, von welchem Typ die Einträge der Felder sind. In der hier untersuchten Lösung liefern Felder immer Einträge vom Java-Typ `Object`. Dieser Typ ist in OQL eigentlich nicht vorgesehen - wird aber benötigt, da sonst die Felder, die von Methoden zurückgeliefert werden, nicht weiter verarbeitet werden können. Der Typ `Object` ist - wie in Java - zu allen anderen Typen kompatibel. Mit Hilfe des Cast-Operators kann dieser Typ in einen spezielleren Typ umgewandelt werden. Liefern also Methoden Kollektionen zurück, so müssen die Elemente der Kollektionen auf den richtigen Typ „gecastet“ werden.

Beispiel:

Angenommen eine Methode liefere eine Liste über Objekten der Klasse `Person`. Dann ist das Ergebnis des Methodenaufrufes vom Typ `List`, intern muß OQL aber immer wissen, um welche Elemente es sich bei der Liste handelt. Da diese Information nicht ersichtlich ist, wurde folgende Lösung vorgenommen: Liefern Methoden Kollektionen als Ergebnis, wird immer angenommen, daß es sich um Kollektionen über dem Typ `Object` handelt. Durch eine explizite Verwendung des Cast-Operators kann der richtige Typ (hier: `Person`) angegeben werden. Es sei hier nochmals bemerkt, daß der Typ `Object` zu allen Java-Klassen und auch zu allen fachlichen Klassen kompatibel ist.

In der OQL-Definition der ODMG wird nicht gesagt, ob `private` oder geschützte Methoden aufgerufen werden können. Diese Möglichkeit, `private` oder geschützte Methoden aufrufen zu können, widerspricht dem Prinzip der Datenkapselung - deshalb wurde in dieser Arbeit das Aufrufen von privaten oder geschützten Methoden innerhalb einer OQL-Anfrage untersagt, d.h. es können nur öffentliche Methoden aufgerufen werden - dabei ist egal, ob diese statisch sind oder nicht.

### 7.4.2.5. Konstruktoren

Mit Hilfe des Typ-Konstruktors kann in OQL eine Instanz einer fachlichen Klasse erzeugt werden. Dazu wird mit Hilfe der Java-Reflection Möglichkeiten der Default-Konstruktor der jeweiligen Klasse aufgerufen. Danach werden die angegebenen Attribute mit den angegebenen Werten gefüllt. Wichtig ist hier zu bemerken, daß in Wirklichkeit nur der Default-Konstruktor aufgerufen wird. Im Anschluß daran werden die angegebenen Attribute mit den übergebenen Werten gefüllt. Dies ist konzeptuell höchst problematisch, da Objekte, durch das Setzen von Attributwerten, Zustände einnehmen können, die sie eigentlich nicht haben dürften. Diese Problematik liegt aber wiederum an der Sprachdefinition von OQL - sinnvollerweise dürften nur die deklarierten Konstruktoren verwendet werden, ein Setzen von beliebigen Attributwerten dürfte nicht erlaubt sein! Die Sprachdefinition der ODMG gibt nicht an, ob die so erzeugten Objekte persistent gespeichert werden oder nicht. In der hier erstellen Arbeit werden Objekte die per Konstruktor erzeugt wurden persistent gespeichert.

### 7.4.2.6. Navigation

Die Interpretation der Navigation zwischen Objekten kann folgendermaßen umgesetzt werden: Da für jede Navigation das Quellobjekt  $Q$  bekannt ist, kann ein SQL-Befehl generiert werden, der das zugehörige Zielobjekt  $Z$  bzw. die zugehörigen Zielobjekte  $Z$  ermittelt:

Beispiel für eine Navigation  $Q.Z$ , bei welcher der Fremdschlüssel in der Quellklasse ist. Der erzeugte SQL-Befehl lautet ( $Z$  sei der Relationenname der Zielklasse):

```
SELECT *
FROM   Z
WHERE  Q.Z-ID = Z.OID
```

wobei für  $Q.Z-ID$  der entsprechende Wert eingetragen wird.

Dieses Verfahren ist extrem Zeitaufwendig, da für jede Navigation zwischen Objekten ein SQL-Befehl erzeugt werden muß. Um diesen Nachteil wettzumachen, wurde in dieser Arbeit das im folgenden Abschnitt beschriebene Verfahren implementiert, mit dem es möglich ist, die Objekt-Navigation effizienter auszuführen.

### 7.4.2.7. Optimierungen

#### 7.4.2.7.1. Optimierung bei der Objekt-Navigation

Bei bereits erwähnt wurde, bestand für die Interpretation von OQL-Anfragen kein Interesse an der Implementierung herkömmlicher Optimierungsmethoden wie Standardisierung, Vereinfachung, Restrukturierung und Transformation.

Von Interesse bei dieser Diplomarbeit ist allerdings, was für Nachteile sich auf das Laufzeitverhalten durch die relationale Datenhaltung ergeben.

Man betrachte die Interpretation der folgenden Anfrage:

```
select a.person.name
from   Adresse a
```

Angenommen, die Datenbank enthält 10.000 Adressen, so wird für jede einzelne Adresse bei der Interpretation ein SQL-Befehl abgeschickt, der die zu der Adresse gehörige Person ermittelt (s.o.), von der dann der Name zurückgeliefert wird. Das bedeutet, für die obige Abfrage müssen 10.001 SQL-Anfragen ausgeführt werden - eine völlig inakzeptable Anzahl!

Damit die Anzahl der zu erzeugenden SQL-Befehle möglichst gering gehalten wird, wurde für die Navigation zwischen Objekten folgendes Lösungsverfahren implementiert:

Findet eine Navigation zwischen Objekten statt, so werden bei der Interpretation alle Objekte sowohl der Quell- als auch der Zielklasse mittels insgesamt zweier SQL-Anfragen ermittelt.

Es werden dann im Speicher die Beziehungen zwischen den Objekten der Quell- und der Zielklasse aufgebaut.

Genauer:

Bereits bei der Typüberprüfung wird ermittelt, auf welchen Pfaden navigiert wird, d.h. es wird gespeichert, von welchen Quellklassen zu welchen Zielklassen navigiert wird. Werden bei der Interpretation in einem Einstiegspunkt alle Objekte einer Quellklasse ermittelt, so werden zu diesen Objekten auch alle Objekte der Zielklassen ermittelt, auf die von der Quellklasse navigiert werden (Navigation kann auch mehrstufig sein). Im Speicher werden dann die Beziehungen zwischen den Objekten aufgebaut. Sind die Objekte der Zielklassen schon im Speicher, so werden diese verwendet und nicht erneut von dem RDBS angefordert. Für die später folgende Navigation muß jetzt kein SQL-Befehl mehr erzeugt werden, sondern nur noch einer Referenz im Speicher gefolgt werden. Der Vorteil dieser Vorgehensweise ist, daß die Anzahl der zu erzeugenden SQL-Anfragen drastisch reduziert werden kann. Im obigen Beispiel werden statt 10.001 nur noch 2 SQL-Anfragen erzeugt, was einen erheblichen Einfluß auf das Laufzeitverhalten hat. Auf den Speicherverbrauch hat diese Vorgehensweise - entgegen schnellen Überlegungen - keine Auswirkungen. Dies liegt daran, daß sowieso alle Objekte für die Interpretation im Speicher gehalten werden müssen (vgl. Abschnitt 7.4.2.3.) - die Attribute welche die Beziehungen (als Referenz) speichern, sind ohne Anwendung dieser Optimierung leer bei Verwendung der Optimierung enthalten sie die entsprechenden Verweise. Es wird also kein zusätzlicher Speicherplatz benötigt.

### 7.4.2.7.2. Optimierung bei der Ermittlung der Daten aus dem RDBS

Wie oben bereits besprochen, werden bei den Einstiegspunkten einfache SQL-Anfragen gestellt, welche die notwendigen Daten aus dem RDBS holen. In manchen Fällen ist es möglich, die Menge der von dem RDBS zurückgelieferten Daten durch zusätzliche Restriktionsbedingungen zu verringern.

Beispiel:

```
select *
from   Person p
where  (p.getAlter()=18) and (p.name like "Duck")
```

Bei dieser Anfrage kann zu der einfachen SQL-Anfrage `SELECT * FROM Person` noch die Restriktionsbedingung `WHERE p.name LIKE "Duck"` angebracht werden. Dadurch wird die von dem Einstiegspunkt zurückgelieferte Menge reduziert, was das Laufzeitverhalten verbessert.

In vielen Fällen kann aber keine Restriktionsbedingung zu der einfachen Anfrage angebracht werden, wie das folgende Beispiel zeigt (Problem: `or` statt `and`):

```
select *
from   Person p
where  (p.getAlter()=18) or (p.name like "Duck")
```

### 7.5. Implementation

Dieser Abschnitt befaßt sich - auf hohem Abstraktionsniveau - damit, wie obiger Entwurf in die Praxis umgesetzt wurde. Der Quellcode für die lexikalische und syntaktische Analyse wurde mit Hilfe des Werkzeugs JavaCC in die Klasse `Parser` generiert. Im folgenden werden deshalb nur noch die Phasen 2 und 3 des groben Entwurfs angesprochen.

#### 7.5.1. Semantische Analyse

Wie bereits im letzten Kapitel angesprochen, bestehen unterschiedliche Konstrukte im Syntaxbaum aus unterschiedlichen Klassen. Allen diesen Klassen ist gemeinsam, daß ihr Name mit dem Präfix `AST` beginnt. Jede Klasse ist dabei Nachfahre der konkreten Klasse `SimpleNode`. Diese Klasse enthält eine abstrakte Methode mit dem Namen `typeCheck`. Diese Methode ist der Kern der semantischen Analyse. Für jedes Konstrukt wird eine eigene Ausprägung dieser Methode implementiert, welche die semantische Analyse des jeweiligen Konstrukts übernimmt. Im Rahmen der semantischen Analyse wird festgestellt, ob ein Konstrukt verwendet wurde, das nicht direkt nach SQL übersetzt werden kann (siehe Kapitel 7.4.1.12). Ist dies der Fall, so wird das private, statische Attribut `must_interpret` auf den Wert `true` gesetzt.

Bestimmte OQL-Konzepte können durch die Verwendung von mehreren, einfacheren OQL-Konstrukten dargestellt werden (vgl. Kapitel 7.4.1.6.1.). Liegt ein solches OQL-Konstrukt vor, so wird der Syntaxbaum verändert, indem der Knoten dieses Konstrukts durch einen semantisch äquivalenten Teilbaum ersetzt wird.

##### 7.5.1.1. Typkonsistenz

Für jedes Konstrukt der Sprache OQL wird in der Methode `typeCheck` die Typ-Konsistenz überprüft. Die Regeln für die Typkonsistenz wurden bereits im Abschnitt über den detaillierten Entwurf sowie im Kapitel 2 dargestellt.

Jede Instanz der Klasse `SimpleNode` enthält ein Attribut, das den Typ angibt, der von dem jeweiligen Konstrukt zurückgeliefert wird. Die Typinformation wird als Zeichenkette gespeichert. Um zu prüfen, ob eine Kollektion oder ein Verbund vorliegt oder, ob Typen kompatibel sind, gibt es in der Klasse `TS` eine Reihe von statischen Methoden. Die Klasse `PI` (steht für POLAR Interface) liefert die Schnittstelle zu dem POLAR<sup>®</sup>-System und enthält Methoden, die angeben, welche Klassen, Attribute und Beziehungen in dem vorliegenden fachlichen Objekt-Modell definiert sind und somit in der OQL-Anfrage verwendet werden können.

##### 7.5.1.2. Sichtbarkeitsregeln

Es wurden nahezu die OQL-Sichtbarkeitsregeln implementiert, die bereits in Kapitel 2 vorgestellt wurden. Nach diesen Regeln ist ein Bezeichner, der im From-Teil deklariert wird, in allen Teilen sowie in allen eingeschachtelten Teilen dieser `Select From Where` Anweisung

sichtbar. Ein Bezeichner, der im Group-Teil deklariert wird, ist in dem Select- und dem Having-Teil sichtbar.

Man betrachte folgende Anfrage:

```
select  a.ort
from    Adresse a
group  by a.strasse
```

Diese Anfrage wäre nach den OQL-Sichtbarkeitsregeln gültig. Sie zeigt aber, daß die Sichtbarkeitsregeln nicht vollständig sind. Denn bei Verwendung eines Group-by Teils dürfen nicht mehr die im From-Teil definierten Iteratorvariablen verwendet werden, sondern nur noch Gruppierungsvariablen.

Was sollte `a.ort` im obigen Beispiel sein? Das Beispiel gruppiert die Straßen; eine anschließende Projektion auf die Orte ist danach nicht mehr möglich, da ja jede Gruppe mehrere Adressen mit evtl. unterschiedlichen Orten enthält - nur die Straßen einer Gruppe sind gleich. Damit obige Anfragen nicht erlaubt sind, mußten die Sichtbarkeitsregeln leicht verändert werden: Wird ein Group by-Teil verwendet, so sind in dem zugehörigen Select-Teil und Having-Teil nur noch die Gruppierungsvariablen sichtbar, unsichtbar sind dagegen die Iteratorvariablen des From-Teils.

Für die korrekte Behandlung von Bezeichnern mußte eine Symboltabelle in der Klasse `SymbolTabelle` implementiert werden. Die Symboltabelle wurde dynamisch, als verkettete Liste bestehend aus Listen, welche die einzelnen Einträge enthalten, implementiert.

### 7.5.2. Umsetzung von OQL- in SQL-Anfragen

Die Umsetzung einer OQL-Anfrage in eine äquivalente SQL-Anfrage wird gestartet, indem die Methode `ToSQL` von dem Wurzelknoten des Syntaxbaumes aufgerufen wird. Diese Methode ruft für jeden Sohn des Wurzelknotens eine Methoden `ToSQL` auf. Um welche Methode `ToSQL` es sich dabei handelt, hängt von dem Knoten selbst - bzw. von dessen Klassentyp ab. Auch die Söhne rufen dann wiederum für alle ihre Söhne eine Methode `ToSQL` auf, usw. Jede Methode `ToSQL` liefert einen Teil der zu erzeugenden SQL-Anfrage in Form einer Zeichenkette zurück. Prinzipiell wird jedes Konstrukt, in der eigens für das Konstrukt vorgesehenen Methode `ToSQL`, nach SQL umgesetzt. In vielen Fällen ist jedoch eine vom Kontext unabhängige Umsetzung nicht möglich. Deshalb wurde die Klasse `UG` (steht für Umgebung) entwickelt, die statische Methoden enthält, mit denen man nach dem aktuellen Umgebungs-Kontext fragen kann. So kann man zum Beispiel fragen, ob der gerade zu bearbeitende `->` Operator in einem Select-Teil vorkommt, usw.

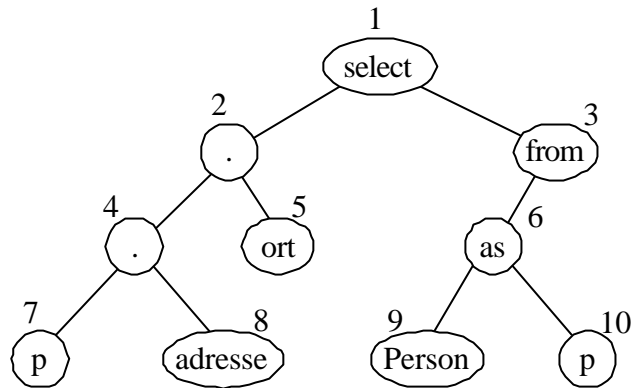
Manche Konstrukte - wie z.B. die Objektnavigationen - können nicht ganz lokal im Syntaxbaum umgesetzt werden. Vielmehr müssen bestimmte Teile der bisher erzeugten Anfrage außerhalb des Syntaxbaumes abgespeichert werden, da sie erst viel weiter oben in dem Syntaxbaum benötigt werden. Durch die dann gewonnenen Informationen (weiter oben im Syntaxbaum) ist in manchen Fällen auch eine Nachbearbeitung der bereits erzeugten SQL-Anfrage notwendig.

## 7. UMSETZUNG VON OQL

Das folgende Beispiel zeigt diese Vorgehensweise, dabei soll die folgende OQL-Anfrage in eine gleichbedeutende SQL-Anfrage übersetzt werden:

```
select p.adresse.ort
from   Person p
```

Diese Anfrage wird im Syntaxbaum folgendermaßen dargestellt:



Die folgende Abbildung zeigt, wie aus den einzelnen Knoten der SQL-Befehl generiert wird (die Zahlen geben an, um welche Knoten es sich handelt):

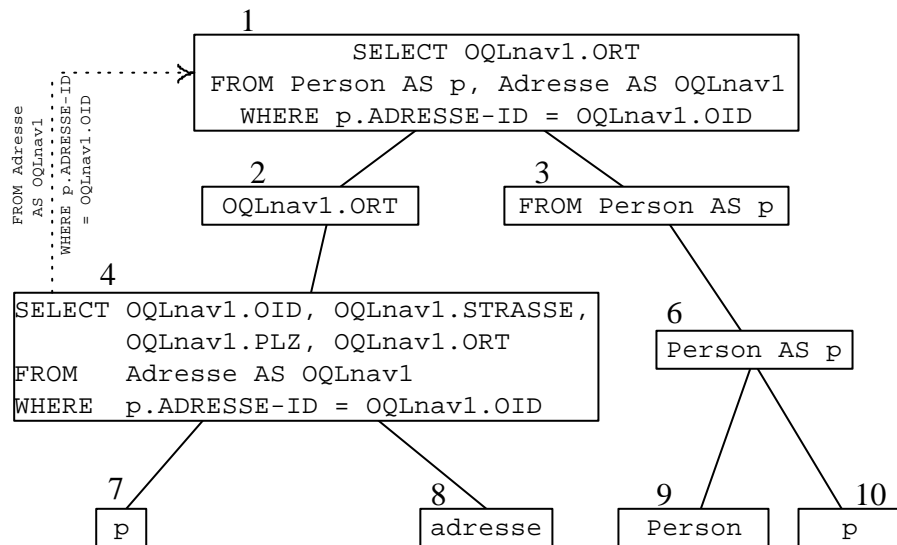


Abbildung 27: Beispiel einer SQL-Generierung

Der gestrichelte Pfeil in obiger Abbildung soll anzeigen, daß hier Informationen - außerhalb der Bearbeitung des Syntaxbaumes - zu weiter oben liegenden Knoten gereicht wird. Genauer:

Der Knoten mit der Nummer 4 speichert in der statischen Klasse `UG` - und somit global - die folgenden Teile einer SQL-Anfrage `"FROM Adresse AS OQLnav1"` und `"WHERE p.adresse-id = OQLnav1.OID"`. Diese Teilanfragen müssen in die umgebene `Select From Where` Anweisung aufgenommen werden. Bei der Bearbeitung des Knotens mit der Nummer 4 ist - für diesen Knoten - aber nicht bekannt, wo sich im Syntaxbaum die umgebene `Select From Where` Anweisung befindet. Deshalb werden diese beiden Anfragen außerhalb des Syntaxbaumes gespeichert und zwar so, daß die umgebene `Select From Where` Anweisung von ihnen Kenntnis nehmen und sie zur Erzeugung der kompletten, umgebenen `Select From Where` Anweisung verwenden kann.

Auch bei der Umsetzung von eingeschachtelten `Select From Where` Anweisungen in dem `From`-Teil einer solchen Anweisung werden Informationen - genau wie oben - außerhalb des Syntaxbaumes nach oben gereicht.

### 7.5.3. Interpretation von OQL-Anfragen

Für die Interpretation der OQL-Anfrage wurde u.a. das `ODMG Java Binding` in dem Package `de.ibl.polar.pos.oqlb` implementiert. Dieses Binding definiert die Interfaces `DCollection`, `DList`, `DArray`, `DSet` und `DBag`. Liefert eine Methode einen der oben aufgezählten Typen oder ein einfaches Literal zurück, so kann das Ergebnis des Methodenaufrufes in der OQL-Anfrage verwendet werden.

Das OQL-Binding sieht bisher noch keine Unterstützung für das Konzept des Verbundes vor. Um dieses Manko zu beheben, wurde das Interface `DStruct` definiert. Liefern Methoden nun ein Ergebnis vom Typ `DStruct` zurück, so kann dieser Methodenaufruf auch in einer OQL-Anfrage verwendet werden.

Für die interne Auswertung der OQL-Anfrage wurden folgenden Datentypen als Klassen implementiert:

<code>IArray</code>	interne Darstellung für ein Feld; implementiert <code>DArray</code>
<code>ISet</code>	interne Darstellung für eine Menge; implementiert <code>DSet</code>
<code>IBag</code>	interne Darstellung für eine Multimenge; implementiert <code>DBag</code>
<code>IList</code>	interne Darstellung für eine Liste; implementiert <code>DList</code>
<code>IAtom</code>	interne Darstellung für Integer, Float, Short, Long, Double, Byte, String, Character, Date, Time, Timestamp und Boolean
<code>IUndefined</code>	interne Darstellung für das Symbol <code>UNDEFINED</code>
<code>INil</code>	interne Darstellung des Literals <code>nil</code>
<code>IStruct</code>	interne Darstellung für einen Verbund; implementiert <code>DStruct</code>
<code>IObject</code>	interne Darstellung für ein fachliches Objekt

Diese Datentypen implementieren alle das Interface `Interpretable`, das die Benennung der Datentypen erlaubt. Durch die Verwendung obiger Datentypen wird die Interpretation einfacher. Dies liegt einerseits daran, daß durch das Interface `Interpretable` Datentypen - wie in der OQL-Anfrage auch - benannt werden können. Andererseits müssen durch die

Zusammenfassung der atomaren Literaltypen in der Klasse `IAtom` nicht so viele Sonderfälle bei der Interpretation beachtet werden.

Die Interpretation wird gestartet, indem die Methode `interpret` von dem Wurzelknoten des Syntaxbaumes an rekursiv aufgerufen wird (wie bei `ToSQL`). Während der Auswertung ist das Ergebnis dieses Methodenaufrufes immer einer der oben angegebenen Typen. Für das Endergebnis wird das Ergebnis in einen Java-Typ umgewandelt, d.h. statt einem `IAtom` wird der von `IAtom` repräsentierte Java-Typ (bspw. ein `Integer`) zurückgeliefert.

Für die Interpretation werden in der Klasse `IP` (steht für `Interpreter`) statische Methoden zur Verfügung gestellt, welche u.a. die Umwandlung zwischen den einzelnen Typen realisieren.

### 7.6. Erweiterungen von OQL

Wie in der Einleitung bereits angesprochen, werden objektorientierte Programmiersprachen hauptsächlich in Verbindung mit relationalen Datenbanksystemen verwendet, wenn der Datenbestand bereits in relationaler Form vorhanden ist. Um Kosten für eine etwaige Umwandlung der Daten in ein anderes Datenmodell zu vermeiden, kann - mit Hilfe von Persistenzframeworks - auf die bestehenden Daten zugegriffen werden, ohne daß mit dem objektorientierten Konzept gebrochen werden muß.

Enthalten aber bereits bestehende relationale Datenbanken Attribute mit den Typen `Time`, `Date` oder `Timestamp`, so können diese in der OQL-Anfrage nicht verwendet werden, da in OQL diese Typen keine atomaren Literale sind. In OQL können `Time`, `Date` und `Timestamp` Typen nur durch Verbunde hergestellt werden.

Damit das Persistenzframework POLAR<sup>®</sup> Anfragen bezüglich aller bereits im RDBS gespeicherten Daten stellen kann, wurde in dieser Arbeit die Sprache OQL um die atomaren Typen `Time`, `Date` und `Timestamp` erweitert.

Die folgenden Beispiele demonstrieren die syntaktische Verwendung dieser drei Typen:

```
select * from Person p where p.geburtsdatum = '1990-01-01'  
  
select * from Person p where p.geburtszeit = '12:12:12'  
  
select * from Person p  
       where p.arbeitsbeginn = '1999-01-01 08:00:00'
```

Für diese Erweiterung von OQL wurde die in Kapitel 6 angegebene Grammatik um Produktionen für die Angabe von `Date`, `Time` und `Timestamp` erweitert. Bei der Typüberprüfung werden die drei Typen als atomare Literale betrachtet.

## 7.7. Schnittstelle zu POLAR

Die Klasse `OQLQuery`, stellt die Schnittstelle der OQL-Unterstützung für POLAR<sup>®</sup> zur Verfügung. Diese Klasse hat - entsprechend den Anforderungen der ODMG - den folgenden Aufbau:

```
class OQLQuery
{
    public OQLQuery();
    public OQLQuery(String query)
        throws QueryInvalidException
    public create(String query)
        throws QueryInvalidException
    public Object execute()
        throws QueryException
    public PObjectContainer search()
        throws QueryException, QueryCannotConvertToSQL
    public Hashtable[] searchFor()
        throws QueryException, QueryCannotConvertToSQL
}
```

Um eine OQL-Anfrage ausführen zu können, muß eine Instanz der Klasse `OQLQuery` erzeugt werden. Dies kann entweder durch den Default-Konstruktor erfolgen, oder mit Hilfe des Konstruktors, der eine Zeichenkette als Parameter verwendet, wobei die Zeichenkette für die auszuführende OQL-Anfrage steht. Eine andere Möglichkeit die OQL-Anfrage zu setzen, besteht darin, den Default-Konstruktor zu verwenden, und mit Hilfe der Funktion `create` eine OQL-Anfrage zu setzen. Läßt sich eine gesetzte OQL-Anfrage nicht compilieren, so wird die Ausnahme `QueryInvalidException` geworfen, die den Fehler genauer spezifiziert.

Die Methode `execute` führt die gesetzte OQL-Anfrage aus. Es sei darauf hingewiesen, daß das Ergebnis von dem Typ `Object` ist, d.h. statt den primitiven Typen (`int`, `short`, `long`, `double`, `float` und `char`) werden die zugehörigen Wrapper-Klassen (`Integer`, `Short`, `Long`, `Double`, `Float` und `Character`) verwendet. Ist das Ergebnis eine Kollektion, so liegt ein im Java Binding (siehe [Catt98]) definierter Typ vor, d.h. entweder ist das Ergebnis vom Typ `DCollection`, `DSet`, `DBag`, `DList` oder vom Typ `DArray`. Ist das Ergebnis ein Verbund, so ist es vom Typ `DStruct`, einem - in dieser Arbeit - definierten Typ, der Verbunde repräsentiert.

Die Methode `execute` kann jede beliebige OQL-Anfrage ausführen. Im besten Fall wird anstelle der OQL-Anfrage eine gleichbedeutende SQL-Anfrage generiert und ausgeführt; im schlechtesten Fall wird die OQL-Anfrage interpretiert. Das Ergebnis der Methode `execute` befindet sich bereits auf dem Niveau der Objekt-Ebene. Findet bei der Interpretation ein Laufzeitfehler statt, so wird die Ausnahme `QueryException` geworfen.

Die Methoden `search` und `searchFor` sind POLAR<sup>®</sup> spezifisch. Sie sind nicht von der ODMG vorgesehen. Die Methode `search` kann eine OQL-Anfrage nur dann ausführen, wenn die OQL-Anfrage in eine einzige SQL-Anfrage übersetzt werden kann, und wenn das Ergebnis der OQL-Anfrage eine Kollektion über Objekten ist. Ist eine OQL-Anfrage gesetzt, die nicht nach SQL übersetzt werden kann, und wird die Methode `search` oder

`searchFor` (s.u.) ausgeführt, so wird eine `QueryCannotConvertToSQL` Ausnahme erzeugt.

Der Aufruf der Methode `search` liefert als Ergebnis eine Kollektion vom Typ `PObjectContainer` zurück. Dieser Datentyp enthält alle Ergebnis-Objekte.

Die Methode `searchFor` liefert als Ergebnis die von SQL zurückgelieferte Ergebnistabelle, repräsentiert als Felder über einer Hash-Tabelle - wobei jedes Feld der Hash-Tabelle einer Zeile in der von dem SQL-Befehl zurückgelieferten Ergebnis-Tabelle entspricht. Die Schlüssel der Hash-Tabelle stellen die Spaltennamen, die Elemente der Hash-Tabelle stellen die zugehörigen Werte dar.

Es sei darauf hingewiesen, daß `searchFor` auf unterster Ebene operiert, d.h. Objekte werden nicht als Objekte sondern als Aufzählungen der einzelnen Spalten zurückgeliefert.

Das einzige Konzept von OQL das in dieser Arbeit nicht umgesetzt wurde, ist das Konzept der benannten Anfragen. Sie werden für das Framework POLAR<sup>®</sup> nicht benötigt und wurden daher weggelassen. Dementsprechend fehlen in der Klasse `OQLQuery` die Methoden zur Definition von benannten Anfragen. Die Umsetzung von benannten Anfragen wäre nicht problematisch - man müßte nur eine spezielle Tabelle anlegen, um die benannten Anfragen als Zeichenketten persistent zu speichern. Die Parameter einer Anfrage könnten als Makro-Expansion gesetzt werden - danach könnte die Anfrage kompiliert und ausgewertet werden, bzw. als Makro-Expansion in einer anderen Anfrage verwendet werden. Abgesehen von dem Konzept der benannten Anfragen wird jedoch die volle OQL-Funktionalität unterstützt.

# 8. Zusammenfassung und Ausblick

Diese Arbeit befaßte sich mit der Umsetzung von OQL-Anfragen an relationale Datenbanksysteme unter Verwendung des Persistenzframeworks POLAR<sup>®</sup>.

Zu Beginn dieser Arbeit wurde in Kapitel 2 der ODMG 2.0 Standard vorgestellt, der u.a. die Sprache OQL definiert. Dabei wurden die einzelnen Konstrukte der Sprache OQL recht detailliert vorgestellt und beschrieben. In Kapitel 2.3.3. wurden Mängel der Sprache OQL aufgezeigt, die bei dem Entwurf und der Implementierung dieser Arbeit zum Vorschein kamen.

Da für die Umsetzung von OQL-Anfragen auf die, von dem Persistenzframework POLAR<sup>®</sup> in dem RDBS gespeicherten, Daten zugegriffen werden muß, war eine Verwendung der Sprache SQL unumgänglich. POLAR<sup>®</sup> kann mit nahezu jedem beliebigen RDBS betrieben werden. Da unterschiedliche RDBS meist unterschiedliche Teilmengen des SQL-Standards implementieren, mußte in Kapitel 3 eine Teilmenge von SQL angegeben werden, die allen von POLAR<sup>®</sup> unterstützten RDBS gemeinsam ist. Diese Teilmenge von SQL bildet die Grundlage für die Umsetzung von OQL-Anfragen in gleichbedeutende SQL-Anfragen.

In Kapitel 4 wurde das Persistenzframework POLAR<sup>®</sup> vorgestellt. Insbesondere wurde angegeben, wie Klassen, Attribute, Strukturhierarchie und Beziehungen auf das RDBS abgebildet werden - diese Informationen bilden das benötigte Grundwissen für die Übersetzung.

In Kapitel 5 wurden die grundlegenden Phasen des Compilerbaus erläutert; notwendig war dies für die Entwicklung eines Compilers, der OQL-Anfragen in adäquate SQL-Anfragen umwandelt. Außerdem wurden die Werkzeuge JavaCC und JJTree erläutert. Mit Hilfe dieser Werkzeuge wurde in Kapitel 6 eine Grammatik für OQL angegeben mit der, unter Verwendung des Werkzeugs JavaCC, ein OQL-Parser generiert werden kann.

Das 7. Kapitel bildet den Kern dieser Arbeit. In diesem Kapitel wurde angegeben, welche grundlegenden Unterschiede zwischen den Sprachen OQL und SQL bestehen. Es wurden Lösungsideen für die Umsetzung von OQL-Anfragen angegeben. Anhand dieser Lösungsideen wurde ein grobes Konzept für die Umsetzung von OQL-Anfragen angegeben: Ist eine Umwandlung der OQL-Anfrage in eine gleichbedeutende SQL-Anfrage möglich, so wird diese Umwandlung mit Hilfe des Compilers vorgenommen, der OQL-Anfragen in gleichbedeutende SQL-Anfragen umsetzt. Ist eine Umwandlung nicht möglich, so wird die OQL-Anfrage interpretiert, dazu werden üblicherweise Daten (Objekte) aus dem RDBS benötigt. Bei der Interpretation werden an den Stellen, an denen in der OQL-Anfrage Einstiegs-Punkte (sog. *Wurzelobjekte*) verwendet wurden, einfache SQL-Anfragen gestellt, die dann die notwendigen Daten aus dem RDBS holen. Mit den zurückgelieferten Daten kann die OQL-Anfrage interpretiert werden. Für die Navigation zwischen Objekten wurde eine Optimierung angegeben, die das Laufzeitverhalten bei der Interpretation verbessert.

Außerdem wurde in diesem Kapitel angegeben, wie die einzelnen Konzepte von OQL nach SQL übersetzt und warum bestimmte Konzepte nicht nach SQL übersetzt werden können.

### *Was sind die Auswirkungen dieser Arbeit?*

Es wurde gezeigt, daß OQL-Anfragen, die komplizierte Navigationspfade enthalten, direkt nach SQL übersetzt werden können. Wird dabei die in Kapitel 7 aufgezeigte Technik verwendet, mit der Navigationspfade aufgelöst werden können, so können Persistenzframeworks - ebenso wie objektorientierte Datenbanksysteme - Anfragen auf Objektebene auswerten, ohne einen Performance-Einbruch zu erleiden. Besonders wichtig ist dies für Anfragen, die nicht durch eine assoziative Suche - die in den meisten Persistenzframeworks implementiert ist - ausgedrückt werden können. So ist es beispielsweise mittels assoziativer Suche unmöglich, alle Personen zu suchen, die keine Adresse haben. Eine solche Anfrage kann leicht in OQL formuliert werden, und mit Hilfe des implementierten Compilers direkt in SQL übersetzt werden.

Auf der anderen Seite wurde gezeigt, daß der Aufruf von Methoden fachlicher Klassen sowie die dadurch ins Spiel kommende Polymorphie nicht ohne Performance Verlust umgesetzt werden kann. Denn für den Aufruf von Methoden fachlicher Klassen muß die gesamte OQL-Anfrage interpretiert werden. Es wurde gezeigt, daß die Interpretation einer OQL-Anfrage im allgemeinen zu Performance Verlusten führt. Um diese Performance-Verluste möglichst gering zu halten, wurde eine Möglichkeit aufgezeigt, wie die Anzahl der zu erzeugenden SQL-Anweisungen bei der Interpretation der Objekt-Navigation verringert werden kann.

### *Was wird sich an den Ergebnissen dieser Arbeit ändern, wenn mächtigere Versionen von SQL verwendet werden - so zum Beispiel SQL-92?*

Bestimmte Konstrukte konnten wegen der eingeschränkten Teilmenge von SQL nicht übersetzt werden, so zum Beispiel die Mengendifferenz, der Mengendurchschnitt und der Zugriff auf Zeichen einer Zeichenkette. Diese OQL-Konstrukte lassen sich unter Verwendung des SQL-92 Standards leicht umsetzen. Auf der anderen Seite gibt es aber Konstrukte, die nicht nach SQL übersetzt werden können, da SQL für relationale Datenbanksysteme konzipiert wurde. Zu nennen ist hier die eingeschachtelte Select From Where Anweisung in dem Select-Teil einer solchen Anfrage. Für dieses Konzept müßte SQL auch Ergebnisse in NF<sup>2</sup>-Form zulassen, was dem relationalen DB-Konzept widerspricht. Auch das Aufrufen von Methoden fachlicher Klassen wird auf Grund des relationalen Konzeptes nicht nach SQL übersetzbar sein.

Kurzum, das Fazit dieser Arbeit ist, daß Persistenzframeworks bei Verwendung von Methodenaufrufen immer den rein objektorientierten Datenbanksystemen in puncto Performance unterlegen sein werden. Hingegen können, mit der in Kapitel 7 aufgezeigten Vorgehensweise, komplizierte OQL-Ausdrücke welche Objekt-navigationen enthalten, direkt nach SQL umgesetzt werden und damit effizient ausgeführt werden. Somit sind Persistenzframeworks den objektorientierten Datenbanksystemen eigentlich nur noch bei der Auswertung von OQL-Anfragen, die Methodenaufrufe enthalten, deutlich unterlegen.

## Anhang A: Beispiele für die Umsetzung

### Beispiel 1:

OQL-Anfrage:

```
select  a.person.name, a.ort
from    Adresse a
where   a.person.vorname like "D*" and a.ort like "E*"
order  by a.person.name
```

generierte SQL-Anfrage:

```
SELECT    OQLnav1x1.NAME AS OQLColumnname,
          a.ORT AS OQLColumnort
FROM      TPERSON AS OQLnav1x1, TADRESSE AS a
WHERE     (a.PERSONID=OQLnav1x1.OID) AND
          ((OQLnav1x1.VORNAME LIKE 'D%') AND
           (a.ORT LIKE 'E%'))
ORDER BY OQLnav1x1.NAME
```

### Beispiel 2:

OQL-Anfrage:

```
select  e
from    Person p, (select pOrt: a.ort, pStrasse:
a.strasse
                    from  p.adresse as a
                    where  a.ort = p.name) as e
where   p.vorname like "D*" and
        e.pStrasse like "N*"
order  by e.pOrt
```

generierte SQL-Anfrage:

```
SELECT    a.ORT AS OQLColumnOrt,
          a.STRASSE AS OQLColumnpStrasse
FROM      TPERSON AS p,TADRESSE AS a
WHERE     (p.OID=a.PERSONID) AND (a.ORT=p.NAME) AND
          ((p.VORNAME LIKE 'D%') AND (a.STRASSE LIKE 'N%'))
ORDER BY a.ORT
```

*Beispiel 3:*

OQL-Anfrage:

```
select  *
from    Person p
where   for all x in p.adresse: x.ort like "E"
```

generierte SQL-Anfrage:

```
SELECT  p.OID,p.VORNAME,p.GEBURTSDATUM,p.NAME,p.ANREDE
FROM    TPERSON AS p
WHERE   ((SELECT COUNT(x.OID)
          FROM  TADRESSE AS x
          WHERE (p.OID=x.PERSONID) AND
                (NOT (x.ORT LIKE 'E%'))
          )=0)
```

*Beispiel 4:*

OQL-Anfrage:

```
select  p
from    Person p, p.adresse as a
where   exists(p.adresse) and
        (a.person.name like "D*") and
        (p.name like "D*")
order  by a.person.name desc, p.vorname asc
```

generierte SQL-Anfrage:

```
SELECT  p.OID,p.VORNAME,p.GEBURTSDATUM,p.NAME,p.ANREDE
FROM    TPERSON AS p,TADRESSE AS a,TPERSON AS OQLnav1x2
WHERE   (p.OID=a.PERSONID) AND
        ((EXISTS(SELECT OQLnav1x1.PERSONID,
                        OQLnav1x1.STRASSE,
                        OQLnav1x1.OID,
                        OQLnav1x1.ORT,
                        OQLnav1x1.POSTLEITZAHL
                    FROM  TADRESSE AS OQLnav1x1
                    WHERE (p.OID=OQLnav1x1.PERSONID)) AND
        (OQLnav1x2.NAME LIKE 'D%')) AND
        (p.NAME LIKE 'D%')) AND
        (a.PERSONID=OQLnav1x2.OID)
ORDER  BY OQLnav1x2.NAME DESC, p.VORNAME ASC
```

## Anhang B: Syntax für JavaCC

Eine Grammatikdatei, die von JavaCC zu bearbeiten ist, muß folgende syntaktische Form haben (aus [JavaCC]):

### **javacc\_input**

```
::= javacc_options
   "PARSER_BEGIN" "(" <IDENTIFIER> ")"
   java_compilation_unit
   "PARSER_END" "(" <IDENTIFIER> ")"
   (production)*
   <EOF>
```

### **javacc\_options**

```
::= [ "options" "{" (option_binding)* "}" ]
```

### **option\_binding**

```
::= "LOOKAHEAD" "=" java_integer_literal ";"
   | "CHOICE_AMBIGUITY_CHECK" "=" java_integer_literal ";"
   | "OTHER_AMBIGUITY_CHECK" "=" java_integer_literal ";"
   | "STATIC" "=" java_boolean_literal ";"
   | "DEBUG_PARSER" "=" java_boolean_literal ";"
   | "DEBUG_LOOKAHEAD" "=" java_boolean_literal ";"
   | "DEBUG_TOKEN_MANAGER" "=" java_boolean_literal ";"
   | "OPTIMIZE_TOKEN_MANAGER" "=" java_boolean_literal ";"
   | "ERROR_REPORTING" "=" java_boolean_literal ";"
   | "JAVA_UNICODE_ESCAPE" "=" java_boolean_literal ";"
   | "UNICODE_INPUT" "=" java_boolean_literal ";"
   | "IGNORE_CASE" "=" java_boolean_literal ";"
   | "USER_TOKEN_MANAGER" "=" java_boolean_literal ";"
   | "USER_CHAR_STREAM" "=" java_boolean_literal ";"
   | "BUILD_PARSER" "=" java_boolean_literal ";"
   | "BUILD_TOKEN_MANAGER" "=" java_boolean_literal ";"
   | "SANITY_CHECK" "=" java_boolean_literal ";"
   | "FORCE_LA_CHECK" "=" java_boolean_literal ";"
   | "COMMON_TOKEN_ACTION" "=" java_boolean_literal ";"
   | "CACHE_TOKENS" "=" java_boolean_literal ";"
   | "OUTPUT_DIRECTORY" "=" java_string_literal ";"
```

### **production**

```
::= javacode_production
   | regular_expr_production
   | bnf_production
   | token_manager_decls
```

### **javacode\_production**

```
::= "JAVACODE"
   java_return_type java_identifier
   "(" java_parameter_list ")"
   java_block
```

### **regular\_expr\_production**

```
::= [lexical_state_list
   regexpr_kind[" " IGNORE_CASE " "]] ":"
```

```
"{" regexpr_spec ( "|" regexpr_spec)* "}"
```

**bnf\_production**

```
::= java_return_type_identifier "(" java_parameter_list ")" ":"
    java_block
    "{" expansion_choices "}"
```

**token\_manager\_decls**

```
::= "TOKEN_MGR_DECLS" ":" java_block
```

**lexical\_state\_list**

```
::= "<" "*" ">" |
    "<" java_identifier ( "," java_identifier )* ">"
```

**regexpr\_kind**

```
::= "TOKEN"
    | "SPECIAL_TOKEN"
    | "SKIP"
    | "MORE"
```

**regexpr\_spec**

```
::= regular_expression [ java_block ] [ ":" java_identifier ]
```

**expansion\_choices**

```
::= expansion ( "|" expansion )*
```

**expansion**

```
::= ( expansion_unit )*
```

**expansion\_unit**

```
::= local_lookahead
    | java_block
    | "(" expansion_choices ")" ["+"|"*"|"?"]
    | "[" expansion_choices "]"
    | [java_assignment_lhs "="] regular_expression
    | [java_assignment_lhs "="] java_identifier
    | "(" java_expression_list ")"
```

**local\_lookahead**

```
::= "LOOKAHEAD" "(" [java_integer_literal]
    ["," ] [expansion_choices] ["," ]
    [{" java_expression "}" ])"
```

**regular\_expression**

```
::= java_string_literal
    | "<" [ [ "#" ] java_identifier ":" ]
    complex_regular_expression_choices ">"
    | "<" java_identifier ">"
    | "<" "EOF" ">"
```

**complex\_regular\_expression\_choices**

```
::= complex_regular_expression
    ( "|" complex_regular_expression )*
```

**complex\_regular\_expression**

```
::= ( complex_regular_expression_unit )*
```

```
complex_regular_expression_unit  
 ::= java_string_literal  
    | "<" java_identifier ">"  
    | character_list  
    | "(" complex_regular_expression_choices ")" ["+"|"*"|"?" ]
```

```
character_list  
 ::= ["~"]["  
    [character_descriptor(", " character_descriptor)*"]"]
```

```
character_descriptor  
 ::= java_string_literal [ "-" java_string_literal ]
```

## Anhang C: Abkürzungsverzeichnis

3GL	Third <u>G</u> eneration <u>L</u> anguages
CORBA	<u>C</u> ommon <u>O</u> bject <u>R</u> equest <u>B</u> roker <u>A</u> rchitecture
DDL	<u>D</u> ata <u>D</u> efinition <u>L</u> anguage
IDL	<u>I</u> nterface <u>D</u> efinition <u>L</u> anguage
JDK	<u>J</u> ava <u>D</u> evelopment <u>K</u> it
JDBC	<u>J</u> ava <u>D</u> ata <u>B</u> ase <u>C</u> onnectivity
ODBC	<u>O</u> pen <u>D</u> ata <u>B</u> ase <u>C</u> onnectivity
ODBS	<u>o</u> bjektorientiertes <u>D</u> aten <u>b</u> ank <u>s</u> ystem
ODMG	<u>O</u> bject <u>D</u> atabase <u>M</u> anagement <u>G</u> roup
ODL	<u>O</u> bject <u>D</u> efinition <u>L</u> anguage
OID	Objekt-Identität
OIF	<u>O</u> bject <u>I</u> nterchange <u>F</u> ormat Language
OMG	<u>O</u> bject <u>M</u> anagement <u>G</u> roup
OO	<u>o</u> bjektorientiert
OQL	<u>O</u> bject <u>Q</u> uery <u>L</u> anguage
POLAR <sup>®</sup>	<u>P</u> ersistent <u>O</u> bject <u>L</u> anguage <u>A</u> bove <u>R</u> elations
POS	<u>P</u> ersistence <u>O</u> bject <u>S</u> ervice
RDBS	<u>r</u> elationales <u>D</u> aten <u>b</u> ank <u>s</u> ystem
SQL	<u>S</u> tructured <u>Q</u> uery <u>L</u> anguage
UML	<u>U</u> nified <u>M</u> odelling <u>L</u> anguage



## Anhang D: Abbildungsverzeichnis

Abbildung 1: Systemarchitektur von POLAR (aus [IBL97], S. 64).....	55
Abbildung 2: Auflösen der Klassenhierarchie im RDBS .....	58
Abbildung 3: Abbildung von 1:1-Beziehungen im RDBS .....	59
Abbildung 4: Abbildung von 1:n-Beziehungen im RDBS .....	59
Abbildung 5: Übersicht über die Werkzeuge und Schnittstellen (aus [IBL97], S. 67) .....	60
Abbildung 6: Architektur des POLAR-Kerns (aus [IBL99], S. 10) .....	61
Abbildung 7: Arbeitsweise eines Compilers (aus [Aho86], S.1).....	64
Abbildung 8: Beispiel für die Arbeitsweise eines Scanners .....	65
Abbildung 9: Beispiel für die Arbeitsweise eines Parsers.....	66
Abbildung 10: Verwendung des Tools JavaCC .....	68
Abbildung 11: Verwendung des Tools JJTree .....	71
Abbildung 12: Umwandlung der OQL-Anfrage in eine gleichbed. SQL-Anfrage .....	82
Abbildung 13: Anfrage in mehrere SQL-Anfragen aufspalten.....	83
Abbildung 14: OQL-Anfrage interpretieren.....	85
Abbildung 15: Allgemeine Select From Where Anweisung im Syntaxbaum.....	90
Abbildung 16: Unäre Operatoren im Syntaxbaum.....	100
Abbildung 17: Binäre Operatoren im Syntaxbaum .....	101
Abbildung 18: Links zeigt den Operator $s[i]$ und rechts den Operator $s[i:j]$ .....	104
Abbildung 19: Existenzquantor und Allquantor im Syntaxbaum .....	105
Abbildung 20: Umwandlung des Allquantors .....	106
Abbildung 21: Umwandlung des Existenzquantors .....	107
Abbildung 22: Umwandlung des unique-Knotens .....	108
Abbildung 23: Aggregatfunktionen im Syntaxbaum .....	108
Abbildung 24: Etwaige Umwandlung von Daten bei der Interpretation .....	116
Abbildung 25: Beispiel einer einfachen Interpretation.....	118
Abbildung 26: Beispiel für die Interpretation einer Anfrage am Syntaxbaum.....	120
Abbildung 27: Beispiel einer SQL-Generierung .....	127

## Anhang E: Literaturverzeichnis

- [Aho86] A.V. Aho, R. Sethi, J.D. Ullmann: *Compilerbau Teil I*, Addison-Wesley, 1986.
- [Aho92] A.V. Aho, R. Sethi, J.D. Ullmann: *Compilerbau Teil II*, Addison-Wesley, 1992.
- [Catt98] R.G.G. Cattell, D.K. Barry: *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers, San Francisco 1998.
- [Coad] P. Coad, E. Yourdon: *Object-Oriented Analysis*, Prentice Hall, 1991
- [Clos91] S. Closs, K. Haag: *Informix-SQL - Sprachbeschreibung*, Hanser Verlag, 1991.
- [Date89] C. J. Date: *A guide to the SQL standard*, 2. Auflage, Addison-Wesley, 1989
- [Date95] C. J. Date: *An Introduction to Database Systems Vol. I*, 6. Auflage, Addison-Wesley, 1995
- [DudInf] H. Engesser, V. Claus: *Duden - Informatik ein Sachlexikon für Studium u. Praxis*, Bibliographisches Institut & F.A. Brockhaus AG, Mannheim, 1988.
- [Härd] T. Härder, E. Rahm: *Datenbanksysteme - Konzepte und Techniken der Implementierung*, Springer-Verlag Berlin-Heidelberg, 1999.
- [Heu92] A. Heuer: *Objektorientierte Datenbanken - Konzepte, Modelle, Systeme*, Addison-Wesley, 1992.
- [Heum] B. Heumesser, A. Ludwig, S. Müller: *Script zum Datenbankpraktikum gehalten im Sommersemester 1999 an der Universität Tübingen*
- [Hop93] J.E. Hopcroft, J.D. Ullmann: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Addison-Wesley, 1993.
- [IBL] Ingenieurbüro Letters GmbH, Homepage: [http:// www.ibl.de](http://www.ibl.de)
- [IBL97] Ingenieurbüro Letters GmbH: *POLAR Whitepaper*, Leinfelden 1997
- [IBL99] Ingenieurbüro Letters GmbH: *Kern/Java: Bedienungsanleitung*, Leinfelden 1999
- [ISO] ISO/IEC 9075:1992(E): *Information technology - Database languages - SQL*, 3. Aufl. 1992
- [itFocus] itFokus Heft 6: *Die objektorientierte Sicht auf Datenbanken SQL geht - OQL kommt?*, it-Verlag, Juni 1999

- [JavaCC] Suntest, Homepage: <http://www.suntest.com/javacc.html>
- [Jän] K. Jänich: *Lineare Algebra*, Springer-Verlag, 5. Aufl. 1996
- [JDK2] Sun Microsystems: Java 2 Dokumentation, Homepage: <http://java.sun.com/products/jdk/1.2/docs/index.html>
- [Jobst91] F. Jobst: *Compilerbau*, Hanser Programmtexte, 1991.
- [Krüger] G. Krüger: *GoTo Java 2*, Addison-Wesley Verlag, 1999
- [Maur92] D. Maurer, R. Wilhelm: *Übersetzerbau - Theorie, Konstruktion, Generierung*, Springer-Verlag, 1992.
- [Myk98] A. Myka: *Folien-Script zur Vorlesung Datenbanksysteme I*, gelesen im Wintersemester 1998/99 an der Universität Tübingen
- [Naka] H. Nakamura and R. E. Johnson: *Adaptive Framework for the REA Accounting Model*, OOPSLA'98 Business Object Workshop IV: <http://jeffsutherland.org/oopsla98/nakamura.html>
- [Neu95] H.A. Neumann: *Objektorientierte Entwicklung von Software-Systemen*, Addison-Wesley, 1995
- [ODMG] Object Data Management Group, Homepage: <http://www.odmg.org>
- [Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modelling and Design*, Prentice Hall, 1991
- [Sch92] U. Schöning: *Theoretische Informatik - kurz gefaßt*, BI-Wissenschaftsverlag, Mannheim, 1992
- [Siel91] B. Sielaff: *Informix-SQL - Kennenlernen*, Hanser Verlag, 1991.
- [Voss94] G. Vossen: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*, Addison-Wesley, 1994.
- [Wan94] G. Wanner: *Entwurf eines auf relationalen Datenbanksystemen basierenden objektorientierten Datenbankmodells für ENFIN-Smalltalk*, Vortrag auf der 'Client/Server ohne Grenzen' in Würzburg, 1994, Nachdruck der IBL GmbH