Entwurf und Implementierung einer Stücklistenverwaltung mittels einer Client/Server-Datenbank

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

in der Studienrichtung
Mechatronik

Eingereicht von

Dipl.-Ing. Christoph Steindl

am Institut für Informatik, Abteilung für Systemsoftware,
der Johannes Kepler Universität Linz
bei **O. Univ.-Prof. Dr. Hanspeter Mössenböck**

Linz, 15. November 1996

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 15. November 1996

Dipl.-Ing. Christoph Steindl

Danksagung

An dieser Stelle möchte ich meinem betreuenden Professor H. Mössenböck und meinen Kollegen J. Templ, M. Hof, M. Knasmüller, C. Mayerhofer und F. Traunmüller danken, die mit mir diskutiert haben und Anregungen eingebracht haben. Auch M. Dätwyler sei gedankt für seine Hinweise. Weiters bin ich meiner Schwester Ulrike sehr dankbar für die sorgsame Korrekturlesearbeit. Auch das gute Arbeitsklima an der Abteilung möchte ich nicht unerwähnt lassen.

Zusammenfassung

In Fabriken werden heute zunehmend Bereiche durch Computereinsatz unterstützt und mittels eines leistungsfähigen Netzwerkes zu einem Rechnerverbund integriert (CIM, *Computer Integrated Manufacturing*). Datenbanken spielen dabei oft eine zentrale Rolle, indem sie alle für die Planungs- und Ausführungsarbeiten relevanten Informationen verwalten und sie Applikationen der unterschiedlichsten Anwendungsbereiche zugänglich machen.

Eine dieser Applikationen ist die Stücklistenverwaltung, bei der es um die mengenmäßige Erfassung aller für ein bestimmtes Erzeugnis nötigen Einzelteile geht. Die Erstellung von Stücklisten ermöglicht es zum Beispiel, mit den Daten der Auftragsplanung (Was soll fabriziert werden?) und den Daten der Entwicklungsabteilung (Was braucht man dazu?) automatisiert Hinweise für die Lagerhaltung zu geben (Wann muß man was zukaufen?).

Im Rahmen dieser Diplomarbeit ist die Anbindung kommerziell verfügbarer Datenbanken an Oberon implementiert worden. Eine benutzerfreundliche Oberfläche für die Administration von Datenbanken und für die Bearbeitung der Daten wurde mit Dialogen gestaltet. Datenbankabfragen können interaktiv in der Abfragesprache SQL formuliert werden.

Abstract

In factories the number of areas supported by the usage of computers is steadily increasing. These computers are integrated by means of a powerful network for cooperation (CIM, *Computer Integrated Manufacturing*). Often databases play a central role by managing all the relevant data for planning and carrying out projects and make the information accessible for applications of all kinds.

One of these applications is the management of parts lists which means the quantitative registration of all individual parts necessary for a certain product. The creation of parts lists enables the system to automatically give advices for the storekeeping (When to buy what?) using the data of the planning of orders (What shall be produced?) and the data of the department of development (What do we need therefore?).

Within the framework of this diploma thesis the connection of commercially available databases to Oberon has been implemented. A user friendly interface for the administration of databases and for the manipulation of the data was designed using dialogues. Database queries can be issued interactively using the query language SQL.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Ziel dieser Diplomarbeit war die Anbindung kommerziell verfügbarer Datenbanken an Oberon, wobei folgende Anforderungen zu erfüllen waren:

- Einfachheit, einfache Benutzung
- flexibler Mechanismus für den Verbindungsaufbau zu Datenbanken
- einfacher Datentransfer zwischen Anwendungsprogramm und Datenbanken
- Interoperabilität
- Unterstützung der Abfragesprache SQL

Um in der Folge den in dieser Diplomarbeit gewählten Weg beurteilen zu können, werden in diesem Kapitel einige Begriffe aus den Bereichen der computerunterstützten Fertigung und der Datenbanken erläutert. Es wird außerdem ein Überblick vermittelt, welche Möglichkeiten es gibt, auf die Daten einer Datenbank von Anwendungsprogrammen aus zuzugreifen.

1.1 Einsatz von Datenbanken in CIM

In Fabriken werden heute zunehmend Bereiche durch Computereinsatz unterstützt und mittels eines leistungsfähigen Netzwerkes zu einem Rechnerverbund integriert (CIM, *Computer Integrated Manufacturing*). Datenbanken spielen dabei oft eine zentrale Rolle, indem sie alle für die Planungs- und Ausführungsarbeiten relevanten Informationen verwalten und sie Applikationen der unterschiedlichsten Anwendungsbereiche zugänglich machen. Die damit erreichbare schnelle Reaktionsfähigkeit zwischen den beteiligten Bereichen, die Datenvollständigkeit und die Mächtigkeit der dezentral arbeitenden, aber miteinander kommunizierenden fertigungstechnischen Regelungskreise bieten die Möglichkeit, die Reaktionszeit zwischen Auftrag und Lieferung bei flexibler Automatisierung der Betriebsmittel zu minimieren.
Man kann die Bereiche, in denen Computer eingesetzt werden, wie folgt aufteilen:

- *CAD (Computer Aided Design)* in der Entwicklung und Konstruktion umfaßt die Bereiche Entwurf, Berechnung und Erprobung von Entwürfen;
- *CAP (Computer Aided Planning)* zur Erzeugung von Teilefertigungs- und Montageanweisungen umfaßt die rechnergestützte Planung der Arbeitsvorgänge und deren Reihenfolgen (Arbeitsfolgen), die Auswahl von Verfahren und Betriebsmitteln, die Erstellung von Daten für die Steuerung der Betriebsmittel sowie die Kalkulation und Simulation;
- *CAM (Computer Aided Manufacturing)* beinhaltet die technische Steuerung und Überwachung der Betriebsmittel bei der Produktion, die Steuerung von Arbeitsmaschinen, Transport- und Lagersystemen und
- *CAQ (Computer Aided Quality Assurance)*, das heißt die Planung und Durchführung der Qualitätssicherung.

Werden nun in allen Bereichen Datenverarbeitungssysteme eingesetzt, so spricht man von *CIM (Computer Integrated Manufacturing)*.

Der Einsatz von Datenbanksystemen in CIM bietet eine Reihe von Vorteilen und auch ein paar Nachteile:

Vorteil Datenunabhängigkeit. Von Datenabhängigkeit spricht man, wenn die Datenorganisation und die Zugriffstechnik von den Anforderungen einer bestimmten Anwendung diktiert werden und das Wissen über diese Organisation und Zugriffstechnik in die Logik der Anwendungsprogramme integriert ist. Der Einsatz eines Datenbanksystems bringt einen Fortschritt in Richtung Datenunabhängigkeit. Die Datenunabhängigkeit bringt eine Entkopplung der Anwender und Anwenderprogramme von der physischen Organisation der Daten und ihren Zugriffspfaden, datenbankinterne Änderungen stören den gewohnten Ablauf der Anwendungsprogramme nicht mehr.

Vorteil Redundanzabbau. Mehrfachspeicherungen von Daten können vermieden werden, wenn die Daten integriert gespeichert und zentral verwaltet werden. Dadurch kommt es zu einer Reduzierung oder zumindest zu einer besseren Kontrolle der Redundanz, was in erster Linie hilft, Speicherplatz zu sparen, sich aber auch günstig auf Datenkonsistenz und Datenintegrität auswirkt.

Vorteil verbesserte Datenkonsistenz und Datenintegrität. Unter Inkonsistenz versteht man die Widerspruchsfreiheit von Daten. Integrität schränkt Konsistenz noch weiter ein: Auch wenn die Daten einander nicht widersprechen (also konsistent sind), können sie falsch sein oder strukturelle Vorgaben des Datenmodells (Integritätsbedingungen) verletzen. Schon allein der Redundanzabbau bewirkt eine Verbesserung der Konsistenz. Die Integrität kann erhöht werden, indem man explizit Integritätsbedingungen formuliert, die dann vom Datenbanksystem automatisch geprüft werden.

Vorteil erhöhte Datensicherheit und besserer Datenschutz. Der Begriff der Datensicherheit bezieht sich auf den Ausfall von Hardware und/oder Software.

Ein Datenbanksystem sollte in der Lage sein, nach denkbaren Ausfällen die betroffenen Datenbanken automatisch in den letzten korrekten Zustand zurückzusetzen. Während die Datensicherheit also auf technische Pannen abzielt, geht es beim Datenschutz um fahrlässiges bis kriminelles menschliches Fehlverhalten. Eine Datenbank muß natürlich gegen unerlaubten Zugriff geschützt werden. Entsprechende Möglichkeiten, wie Vergabe und Entzug der Zugriffsrechte, sollte jedes Datenbanksystem unterstützen. Integrierte Speicherung, zentrale Verwaltung und die Kontrolle durch den Datenbankadministrator bieten erhöhte Datensicherheit und besseren Datenschutz als traditionelle Dateiverarbeitung.

Nachteil Ressourcenbedarf. Datenbanksysteme sind umfangreiche und komplexe Softwaresysteme, für die entsprechende Hardware-Kapazitäten bereitgestellt werden müssen.

Vor- und Nachteil Datenbanksprache. Der Anwendungsprogrammierer muß die entsprechende Datenbanksprache beherrschen. Durch eine Standardisierung dieser Datenbanksprache wird dieser Nachteil zu einem Vorteil, da der Anwendungsprogrammierer mit seinem Fachwissen problemlos auf neue Datenbanksysteme wechseln kann. Außerdem können dann bestehende Programme leicht übernommen werden.

Nachteil Lizenzgebühren. In der Regel fallen nicht unerhebliche Lizenzgebühren für die Nutzung von Datenbanksystemen an.

Stücklisten

Die Verwaltung von Stücklisten ist eine der vielen Anwendungen im Rechnerverbund. Bei ihr geht es um die mengenmäßige Erfassung aller für ein bestimmtes Erzeugnis nötigen Einzelteile. Die Erstellung von Stücklisten ermöglicht es zum Beispiel, mit den Daten der Auftragsplanung (Was soll fabriziert werden?) und den Daten des CAD (Was braucht man dazu?) automatisiert Hinweise für die Lagerhaltung zu geben (Wann muß man was zukaufen?).

Stücklisten enthalten üblicherweise Positionsnummern, Menge, Mengeneinheit, Benennung der Gruppe oder des Teils (einschließlich Normteile, Fremdteile und Hilfsstoffe), Sachnummer und/oder Norm-Kurzbezeichnung zur Identifikation sowie sonstige Bemerkungen. Stücklisten-Einträge bestehen aus einem Schriftfeld und einem Stücklistenfeld, deren Aufbau in DIN 6771, Teil 1 und 2 festgelegt sind. Man unterscheidet folgende Arten von Stücklisten [Dubbel]:

- *Mengenübersichts-Stücklisten* enthalten für das Erzeugnis nur die Einzelteile mit ihren Mengenangaben. Mehrfach vorkommende Teile erscheinen nur einmal. Funktions- und fertigungsorientierte Gruppen sind nicht zu erkennen. Diese einfachste Form einer Stückliste reicht für Erzeugnisse mit nur wenigen Fertigungsstufen aus.

- *Struktur-Stücklisten* geben die Erzeugnisstruktur mit allen Baugruppen und Teilen wieder. Die Gliederung der Gruppen und Teile entspricht in der Regel dem Fertigungsablauf. Die Mengenangaben beziehen sich auf das im Stücklistenkopf beschriebene Erzeugnis. Die Gesamtstruktur eines Erzeugnisses bzw. einer Gruppe ist erkennbar, allerdings werden Stücklisten mit vielen Positionsnummern unübersichtlich, vor allem wenn eine Reihe von Wiederholgruppen an jeweils verschiedenen Stellen wiederkehrt. Dadurch ergeben sich auch Nachteile im Änderungsdienst.

- *Baukasten-Stücklisten* umfassen zusammengehörende Gruppen und Teile, ohne auf ein bestimmtes Erzeugnis Bezug zu nehmen. Die Mengenangaben beziehen sich auf die im Kopf genannte Baugruppe. Mehrere solche Baukasten-Stücklisten können (gegebenenfalls mit anderen Stücklisten) zu einem Stücklistensatz eines Erzeugnisses zusammengestellt werden. Ihre Verwendung empfiehlt sich dort, wo bei einem größeren Erzeugnisspektrum Baugruppen lagermäßig geführt und als Wiederholgruppen in größeren Stückzahlen gefertigt werden.

1.2 Relationen und Relationenalgebra

Die Implementierung einer Stücklistenverwaltung erfolgt am besten mit einer Datenbank unter Beachtung existierender Datenbank-Standards. Ein weit verbreiteter Standard sind *relationale Datenbanken*. Für ein gutes Verständis relationaler Datenbanken ist allerdings auch ein Verständnis des zu Grunde liegenden Datenmodells – der Relationen – notwendig. Das Relationenmodell geht auf Codd zurück und wurde in [Codd70] zum ersten Mal vorgestellt[1].

Im relationalen Datenmodell hat man *Domänen* als benannte Mengen von atomaren (= skalaren, elementaren, nicht strukturierten) Werten. Domänen entsprechen also im wesentlichen den Datentypen höherer Programmiersprachen mit impliziter Festlegung der auf diesen Datentypen erlaubten Operationen. Eine *Relation* ist eine Untermenge des kartesischen Produkts von einer oder mehreren Domänen. Die einzelnen Elemente einer Relation werden *Tupel* genannt, die selbst wieder aus *Attributen* bestehen. Relationen haben folgende Eigenschaften:

- Es kann niemals mehrfach auftretende gleiche Tupel in einer Relation geben.
- Die Tupel einer Relation sind nicht geordnet.
- Auch die Attribute einer Relation sind nicht geordnet.
- Alle Attributwerte sind atomar.

Wegen der Definition einer Relation als Menge muß jedes Tupel dieser Menge eindeutig identifizierbar sein. Man kann deswegen einen *Schlüsselkandidaten* wie folgt definieren: Sei R eine Relation mit den Attributen $A_1, A_2, \ldots A_n$. Die Menge

[1] Codd erhielt für die Entwicklung dieses Modells im Jahre 1981 den ACM Turing Award.

$K = \{A_{i1}, A_{i1}, \ldots A_{im}\}$, $1 \leq m \leq n$, heißt *Schlüsselkandidat* von R genau dann, wenn unabhängig vom Zeitpunkt (also für die gesamte Lebensdauer der Relation) gilt:

1. *Eindeutigkeit*: Zu keinem Zeitpunkt gibt es zwei Tupel mit demselben Wert für K (das heißt mit demselben Wert für A_{i1}, demselben Wert für A_{i2},... und demselben Wert für A_{im}).

2. *Minimalität*: Keines der Attribute $A_{i1}, A_{i2}, \ldots A_{im}$ kann weggelassen werden, ohne die Eindeutigkeit von K zu verlieren.

Man kann für jede Relation R einen (beliebigen) Schlüsselkandidaten als *Primärschlüssel (primary key)* festlegen. Die übrigen Schlüsselkandidaten heißen *Alternativschlüssel (alternate keys)*.

Ein *Fremdschlüssel* ist eine (möglicherweise nur einelementige) Menge von Attributen einer Relation R, die in einer anderen Relation R_1 als Primärschlüssel auftritt. R und R_1 sind nicht notwendigerweise voneinander verschieden.

Primärschlüssel kann man im Relationenmodell als *symbolische Tupeladressen* auffassen. Entsprechend ist ein *Fremdschlüssel* ein *symbolischer Zeiger* auf ein bestimmtes Tupel einer Relation.

Im Relationenmodell können *Beziehungen* zwischen Relationen nur durch Fremdschlüssel dargestellt werden. Mit den folgenden Beziehungen zwischen Relationen kann man Bedingungen für die Integrität der Datenbank aufstellen:

Entitätsintegrität (entity integrity): Für jede definierte Relation muß ein Primärschlüssel festgelegt sein. Kein Attribut, das zum Primärschlüssel einer Relation gehört, darf *Nullwerte* annehmen. Dabei versteht man unter einem Nullwert eine spezielle Markierung für unbekannte Werte. Diese Markierung darf nicht zum Wertevorrat der entsprechenden Domäne gehören.

Referentielle Integrität (referential integrity): Sei R_2 eine Relation, die mit einem Fremdschlüssel F auf eine Relation R_1 mit Primärschlüssel P zeigt. Für jeden in R_2 auftretenden Wert von F (= symbolischer Zeiger) muß dann gelten:

1. Er enthält entweder in keiner seiner Komponenten einen Nullwert (vollständig definiert) oder alle seine Komponenten sind mit Nullwerten markiert (vollständig annulliert).

2. Wenn er vollständig definiert ist, muß in R_1 ein Tupel existieren, das diesen Wert als P-Wert aufweist (keine *dangling pointers*).

In der *Relationenalgebra* sind die folgenden fünf Grundoperationen [Ull88] definiert:

1. *Vereinigung*. Die Vereinigung von zwei Relationen R und S (die Relationen müssen natürlich denselben Aufbau, d.h. gleiche Attributmengen, haben), geschrieben als $R \cup S$, ist die Menge der Tupel, die in R oder in S oder in beiden Relationen enthalten sind.

2. *Differenz*. Die Differenz von zwei Relationen R und S (die Relationen müssen natürlich kompatibel sein), geschrieben als $R-S$, ist die Menge der Tupel von R, die nicht auch in S enthalten sind.

3. *Kartesisches Produkt*. Seien R und S Relationen mit k_1 und k_2 Attributen. Dann ist $R \times S$, das kartesische Produkt der Relationen R und S, die Menge aller möglichen (k_1+k_2)-Tupel, wobei die ersten k_1 Attribute von einem Tupel der Relation R stammen und die letzten k_2 Attribute von einem Tupel der Relation S.

4. *Projektion*. Die Projektion extrahiert bestimmte Attribute aus einer Relation und bildet somit eine *vertikale Teilrelation*. Sei R eine Relation mit k Attributen, dann ist $\pi_{i1,i2,\dots im}(R)$ mit paarweise unterschiedlichen i_j (j im Bereich 1 bis m) die Projektion der Relation R auf die Komponenten $i_1, i_2, \dots\, i_m$, das heißt die Menge der m-Tupel $a_1a_2\cdots\, a_m$, sodaß es ein k-Tupel $b_1b_2\cdots\, b_k$ in der Relation R gibt, für das $a_j = b_i$ für $1 \leq j \leq m$ und $1 \leq i \leq k$ gilt.

5. *Selektion*. Die Selektion extrahiert aus einer Relation bestimmte Tupel und bildet somit eine *horizontale Teilrelation*. Die Selektion $\sigma_F(R)$ ist die Menge der Tupel der Relation R, die die Bedingung F erfüllen.

1.2.1 Normalformen

Es gibt gewisse Richtlinien, die man bei der Datenmodellierung befolgen sollte, um Redundanzen zu vermeiden. Am einfachsten erreicht man das, wenn man überprüft, ob die Relationen in *Normalform* sind.

Man definiert die folgenden Arten von Normalformen (siehe [Heu95]):

1. Normalform

Die Attribute der Relation müssen atomar sein (ist eine schon vorausgesetze Eigenschaft von Relationen). Strukturierte Attribute (wie Adresse) müssen aufgeteilt werden in ihre Teilattribute (z.B. in PLZ, Ort, Straße und Hausnummer). Relationen in erster Normalform werden auch oft als flache Relationen bezeichnet. Aufgrund von funktionalen Abhängigkeiten (PLZ bestimmt Ort) ergeben sich in 1NF-Relationen Redundanzen.

2. Normalform

Die zweite Normalform vermeidet partielle funktionale Abhängigkeiten (diese bewirken Redundanzen). Eine partielle funktionale Abhängigkeit besteht, wenn Attribute (die nicht Schlüsselkandidaten sind) funktional schon von einem Teil des Schlüssels abhängen (siehe Abbildung 1.1). Die zweite Normalform kann durch Elimination der abhängigen Attribute und Auslagerung in eine eigene Relation erreicht werden.

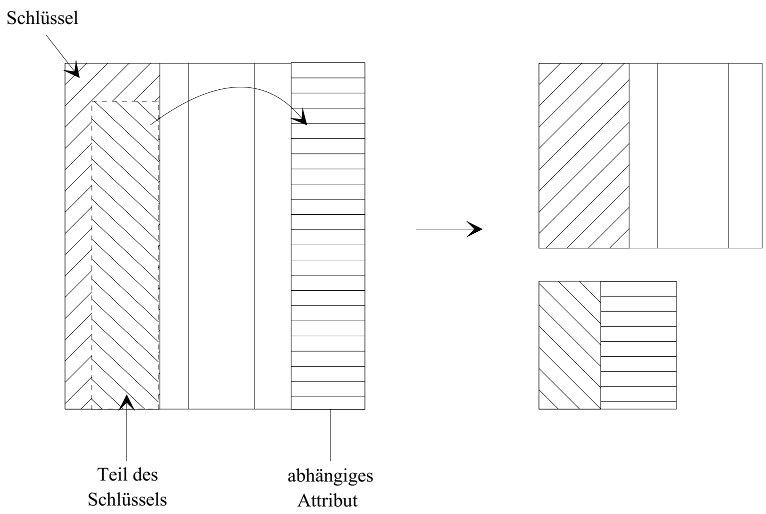

Abbildung 1.1: 2. Normalform

3. Normalform

Die dritte Normalform löst transitive Abhängigkeiten auf. Geht man von einem Schlüssel aus, der eine Attributmenge bestimmt, die wiederum ein abhängiges Attribut bestimmt, so liegt eine transitive Abhängigkeit vor (siehe Abbildung 1.2). Zur Beseitigung kann man das transitiv abhängige Attribut in eine neue Relation kopieren (gemeinsam mit der bestimmenden Attributmenge) und aus der ursprünglichen Relation entfernen.

1.3 SQL

Das relationale Datenmodell verfügt mit SQL (Structured Query Language) über eine mächtige Datenbanksprache. Die Entwicklung von SQL begann in den frühen 70er Jahren und wurde im IBM Forschungslabor in San José durchgeführt. Die erste Sprachversion hieß noch SEQUEL (für Structured English Query Language). 1983 begann das American National Standards Institute (ANSI) mit der Ausarbeitung eines Standards für relationale Datenbanksprachen. Der Abschlußbericht wurde als American National Standard Database Language - SQL (ANSI X3.135-1986) verabschiedet. Dieser Standard wurde 1987 auch von der International Standards Organisation (ISO) als ISO-Standard übernommen (ISO 9075 Database Language SQL). In den Jahren 1986 beziehungsweise 1989 gab es weitere Standarddokumente, für die sich die Begriffe SQL-86, SQL-89 und SQL1 als Überbegriff entwickelt haben. 1992 kam eine überarbeitete Version als SQL-92 oder SQL2 heraus, das SQL2-Standarddokument umfaßt bereits mehr als 600 Seiten gegenüber den 120 Seiten des SQL-89 Dokuments. SQL2 gliedert sich in drei Sprachebenen: Entry SQL, Intermediate SQL und Full SQL. Dadurch konnten die Anbieter von Datenbanksystemen

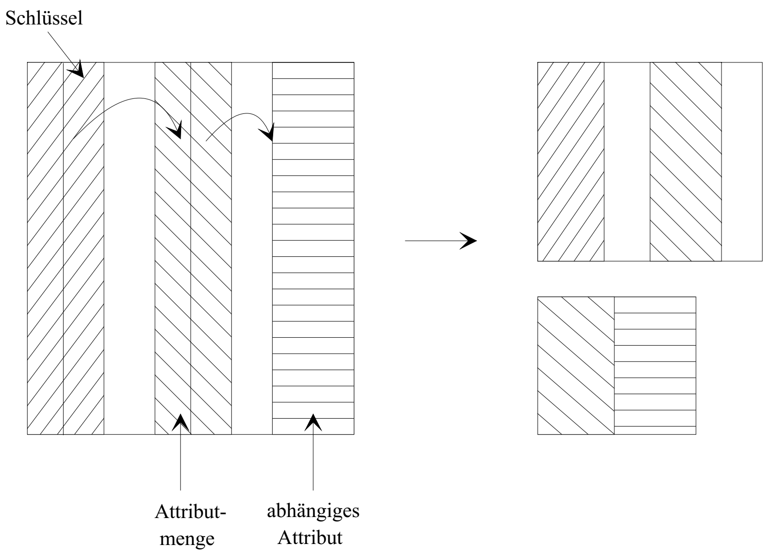

Abbildung 1.2: 3. Normalform

wenigstens rasch Entry SQL implementieren und erst in späteren Versionen den gesamten Sprachumfang abdecken. Zur Zeit wird von den Standardisierungsorganisationen bereits an der dritten Generation von SQL gearbeitet.

1.3.1 Datendefinitionsbefehle von SQL

SQL bietet Befehle für die Datendefinition an (DDL statements, data definition language). Mit diesen Befehlen legt man zum Beispiel die Domänen der einzelnen Attribute der Relationen fest. Weil man Relationen in einfacher Weise auch als Tabellen darstellen kann, spricht SQL in der Regel von Tabellen (tables). Man darf sich aber dabei nicht verleiten lassen zu glauben, daß die Anordnung der einzelnen Tupel innerhalb der Tabelle irgendeine Bedeutung hätte.

Im folgenden wird eine kurze Übersicht über die einzelnen Datendefinitionsbefehle gegeben.

- CREATE TABLE tableName "(" colElem {"," colElem} ")"
  colElem = colDefinition | tableConstraint.
  colDefinition = colIdent dataType [DEFAULT defValue] {colConstraint}.
  defValue = literal | NULL | USER.
  colConstraint = NOT NULL | UNIQUE | PRIMARY KEY
  | REFERENCES tableName referencedColumns
  | CHECK "(" searchCondition ")".
  tableConstraint = UNIQUE "(" colIdent {"," colIdent} ")"
  | PRIMARY KEY "(" colIdent {"," colIdent} ")"
  | CHECK "(" searchCondition ")"

| FOREIGN KEY referencingCols REFERENCES
referencedTable referencedColumns.

CREATE TABLE dient zum Erstellen einer neuen Relation (Tabelle) mit den angegebenen Attributen (Spalten). Zusätzlich kann man Integritätsbedingungen definieren. Im folgenden Beispiel ist das Attribut no jeweils der Primärschlüssel der Relationen, weiters muß für jeden Eintrag in der Relation Addresses ein Eintrag in der Relation Friends mit entsprechendem Wert von no existieren (referentielle Integrität).
Beispiel: CREATE TABLE Friends (no INTEGER NOT NULL PRIMARY KEY,
firstName CHAR(32), lastName CHAR(32) NOT NULL)
CREATE TABLE Addresses (no INTEGER NOT NULL PRIMARY KEY
REFERENCES Friends no, street CHAR(50), town CHAR(20),
zipCode INTEGER)

- ALTER TABLE tableName (ADD colIdent dataType
| ADD "("colIdent dataType {"," colIdent dataType}")"
| DROP [COLUMN] colIdent [CASCADE | RESTRICT])

  ALTER TABLE dient zum Hinzufügen und Entfernen von Attributen (Spalten) zu beziehungsweise von einer Relation (Tabelle). Die neuen Attribute in bestehenden Tupeln erhalten entweder Nullwerte oder die Werte, die man bei der Tabellendefinition voreingestellt hat (DEFAULT defValue).
  Beispiel: ALTER TABLE Friends ADD (zipCode CHAR(5), town CHAR(50))

- DROP TABLE tableName [CASCADE | RESTRICT]

  DROP TABLE löscht die angegebene Relation (Tabelle).
  Beispiel: DROP TABLE Friends

- CREATE [UNIQUE] INDEX indexName ON tableName
colIdent [ASC | DESC] {"," colIdent [ASC | DESC]}

  CREATE INDEX erstellt einen neuen Index über Spalten einer Tabelle. Mittels eines Indizes können gewisse Abfragen wesentlich schneller abgearbeitet werden.
  Beispiel: CREATE INDEX idx ON Friends (lastName, firstName)

- DROP INDEX indexName

  DROP INDEX löscht den angegebenen Index.
  Beispiel: DROP INDEX idx

- CREATE VIEW viewName ["(" colIdent {"," colIdent} ")"] AS query
  query = SELECT [ALL | DISTINCT] selectList FROM tableList
  [WHERE searchCondition]
  [GROUP BY colName {"," colName}]
  [HAVING searchCondition].

  CREATE VIEW erstellt eine neue logische Sicht auf eine bestehende Tabelle oder auf das Ergebnis einer Abfrage. Eine Erklärung der Syntax von Abfragen folgt weiter hinten. Der Stern im folgenden Beispiel ist synonym mit dem Schlüsselwort ALL.
  Beispiel: CREATE VIEW FriendsInLinz AS SELECT * FROM Friends
  WHERE town = 'LINZ'

- DROP VIEW viewName [CASCADE | RESTRICT]

  DROP VIEW löscht eine logische Sicht.
  Beispiel: DROP VIEW FriendsInLinz

- GRANT (ALL | grantPrivilege {"," grantPrivilege} ON tableName)
  TO (PUBLIC | userName {, userName})
  grantPrivilege = DELETE | INSERT | SELECT
  | UPDATE [colIdent {"," colIdent}]
  | REFERENCES [colIdent {"," colIdent}].

  GRANT dient zum Erteilen von Rechten auf Relationen.
  Beispiel: GRANT ALL ON Friends TO PUBLIC

- REVOKE (ALL | revokePrivilege {"," revokePrivilege} ON tableName)
  FROM (PUBLIC | userName {"," userName}) [CASCADE | RESTRICT]
  revokePrivilege = DELETE | INSERT | SELECT | UPDATE
  | REFERENCES.

  REVOKE widerruft Rechte auf Relationen.
  Beispiel: REVOKE ALL ON Friends FROM PUBLIC

1.3.2 Datenmanipulationsbefehle von SQL

Die Befehle zur Manipulation der Daten in SQL (DML statements, data manipulation language) beschränken sich auf die vier Operationen Einfügen, Ändern, Entfernen und Auswählen.

- INSERT INTO tableName [colIdent {"," colIdent}]
  (query | VALUES "("insertValue {"," insertValue}")")

INSERT fügt ein neues Werte-Tupel in eine Relation ein.
Beispiel: INSERT INTO Friends VALUES ('Christoph', 'Steindl', '4040', 'Linz')

- UPDATE tableName SET colIdent "=" (expr | NULL)
  {"," colIdent "=" (expr | NULL)}
  (WHERE searchCondition | CURRENT OF cursorName)

  UPDATE ändert entweder alle Tupel einer Relation, die die Bedingung erfüllen, oder dasjenige Tupel, über dem der Cursor positioniert ist.
  Beispiel: UPDATE Friends SET town = 'Linz' WHERE lastName = 'Steindl'

- DELETE FROM tableName (WHERE searchCondition
  | CURRENT OF cursorName)

  DELETE entfernt entweder alle Tupel einer Relation, die die Bedingung erfüllen, oder dasjenige Tupel, über dem der Cursor positioniert ist.
  Beispiel: DELETE FROM Friends WHERE town = 'Linz'

- SELECT [ALL | DISTINCT] selectList FROM tableList
  [WHERE searchCondition]
  [GROUP BY colName {, colName}]
  [HAVING searchCondition]
  {UNION [ALL] selectStatement}
  [ORDER BY sortSpecification {, sortSpecification}
  SELECT [ALL | DISTINCT] selectList FROM tableList
  [WHERE searchCondition]
  FOR UPDATE OF colName {, colName}

  SELECT wählt von den Tupeln einer oder mehrerer Relationen die aus, die bestimmte Bedingungen erfüllen. In der zweiten Variante bereitet man die ausgewählten Tupel für eine Änderung vor.

Die Abarbeitung einer SELECT-Anweisung soll hier kurz erläutert werden:

1. FROM Klausel: Zuerst wird die FROM Klausel bearbeitet. Das Ergebnis dieses Schrittes ist das kartesische Produkt der angegebenen Tabellen.

2. WHERE Klausel: Im zweiten Schritt werden aus der entstandenen Tabelle alle Tupel (Zeilen) entfernt, die die Suchbedingung nicht erfüllen.

3. GROUP BY Klausel: In der GROUP BY Klausel wird ein Gruppierungsschlüssel definiert. Für jeden in der entstandenen Tabelle auftretenden Wert des Gruppierungsschlüssels wird eine Gruppe gebildet. Eine Gruppe besteht dabei aus einem oder mehreren Tupeln (Zeilen).

4. HAVING Klausel: Die HAVING Klausel ist das gruppenorientierte Pendant zur zeilenorientierten WHERE Klausel, es werden diejenigen Gruppen entfernt, die die Suchbedingung nicht erfüllen. Ist eine HAVING Klausel bei fehlender GROUP BY Klausel vorhanden, so wird die gesamte Relation (Tabelle) als eine einzige Gruppe betrachtet.

5. SELECT Liste: Erst nach der Auswertung des Tabellenausdrucks werden die Angaben der SELECT Liste ausgewertet. Dabei wird eine Projektion auf die ausgewählten Attribute (Spalten) vorgenommen, wobei auch Wertausdrücke (wie SUM, MAX und MIN) ausgewertet und deren Ergebnis eingefügt werden. Wurde das Schlüsselwort DISTINCT angegeben, werden anschließend auch etwaige Duplikate aus der Ergebnisrelation entfernt.

Beispiel: `SELECT * FROM Friends GROUP BY town`

Die hier gezeigten SQL-Anweisungen stimmen in ihrer genauen Formulierung vielleicht nicht im Detail mit denen der SQL2-Standarddokumente überein, sind jedoch so angegeben, wie sie ODBC-Datenbanken erwarten und interpretieren können [ODBC94]. Da hier nur die generelle Syntax von SQL-Anweisungen wiederholt werden soll, sollten die (feinen) Unterschiede nicht stören.

1.3.3 PL/SQL

PL/SQL [PLSQL] ist eine prozedurale Erweiterung der deklarativen Abfragesprache SQL, es gibt zum Beispiel Schleifen und bedingte Verzweigungen, man hat aber keine Berechnungsvollständigkeit und keine Modularisierung.

PL/SQL erlaubt die Ausführung komplexer datenbankorientierter Operationen innerhalb des Datenbank-Servers, wobei die Kontrolle nicht nach jeder SQL-Anweisung an das Anwendungsprogramm zurückgegeben werden muß. Außerdem hat man eine einheitliche Sprache für alle Anwendungsprogramme und für die Administration des Datenbanksystems.

PL/SQL ist eine blockorientierte Sprache, wobei jeder Block aus einem optionalen Deklarationsteil (Variablen- und Konstantendeklarationen, explizite Cursor-Definitionen und benutzerdefinierte Ausnahmen), einem Ausführungsteil (dem eigentlichen Programm) und einem optionalen Fehler- und Ausnahmebehandlungsteil besteht.

Ein Cursor ist ein Verweis auf den privaten Arbeitsbereich einer SQL-Anweisung (private SQL area). Das Cursorkonzept ist eine Erweiterung von SQL, die in PL/SQL eingeführt wurde, um die zeilenweise Abarbeitung des Abfrageergebnisses zu ermöglichen. Ein Cursor kann implizit deklariert sein für DDL- und DML-Anweisungen einschließlich SELECT, die nur eine einzige Zeile liefern, oder explizit für mengenorientierte Abfragen, die mehr als eine Zeile liefern. Mit Aufrufen der Fetch-Operation wird der Cursor Datensatz für Datensatz über die Ergebnismenge bewegt. Man kann immer nur auf die Werte der Attribute des aktuellen Datensatzes zugreifen und hat folgende Möglichkeiten, den Status eines Cursors zu überprüfen: %FOUND ist TRUE, wenn die letzte FETCH-Operation einen Datensatz gefunden hat (analog dazu ist %NOTFOUND TRUE, wenn die letzte FETCH-Operation keinen Datensatz gefunden hat), %ROWCOUNT liefert die Anzahl der bereits gefundenen Datensätze und %ISOPEN ist TRUE, wenn der Cursor geöffnet ist.

Wie schon vorher erwähnt, gibt es in PL/SQL Kontrollstrukturen:

Unbedingte Schleife

```
LOOP
  ...
  [EXIT [WHEN condition]; ]
  ...
END LOOP;
```

FOR-Schleife

```
FOR counter IN [reverse] start .. end LOOP
  ...
END LOOP;
```

FOR-Cursor-Schleife

```
DECLARE
  CURSOR curFor IS SELECT * FROM Friends ORDER BY lastName;
  ...
BEGIN
  FOR c1Rec IN curFor LOOP
    ...
    count := curFor%ROWCOUNT;
    ...
  END LOOP;
```

Bei einer FOR-Cursor-Schleife wird implizit eine Datensatzstruktur gemäß der Cursor-Deklaration erstellt, der Cursor geöffnet, die Datensätze (einer nach dem anderen) aus der Datenbank in die Struktur übertragen und der Cursor nach Beenden der Schleife wieder geschlossen.

Ein kurzes Beispiel soll einen Eindruck geben, wie PL/SQL-Programme aussehen können.

```
DECLARE
  CURSOR c1 IS SELECT * FROM Friends ORDER BY lastName;
  c1Rec c1%ROWTYPE;
BEGIN
  ...
  OPEN c1;
  ...
```

```
  LOOP
    FETCH c1 INTO c1Rec;
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
  ...
  count := c1%ROWCOUNT;
  CLOSE c1;
  ...
END
```

1.3.4 Embedded SQL

Das ursprüngliche Ziel der Entwickler von Embedded SQL war der Entwurf einer Sprache, die sowohl interaktiv nutzbar als auch in 3. Generationssprachen einbettbar sein sollte. Mit Embedded SQL erreicht man eine Datenbankanbindung bestehender 3. Generationssprachen. Embedded SQL erlaubt die Spezifikation von SQL-Anweisungen zur Laufzeit und bietet Kompatibilität zu anderen relationalen Datenbanksystemen durch den ANSI-Standard. Im Gegensatz zu PL/SQL hat man auch die Möglichkeit, die Ergebnisse in ansprechender Form auszugeben. Das entstehende Programm ist allerdings eng an die Datenbankstruktur gebunden.

In Embedded SQL werden SQL Befehle in normale Programme (Hostprogramme, geschrieben in einer beliebigen Hostsprache wie COBOL, Pascal oder C) eingefügt. Diese Befehle beginnen und enden dabei mit speziellen Anfangs- und Endsequenzen (zum Beispiel Beginn mit „EXEC SQL“ und Ende mit „;“). Ein Precompiler übersetzt die SQL Befehle in äquivalenten Quellcode, sodaß das Programm zum Beispiel nur mehr C-Code enthält und anschließend mit einem normalen C-Compiler übersetzt werden kann. Folgende Anweisungen sind in Embedded SQL erlaubt:

- ausführbare SQL-Anweisungen
  - zur Herstellung der Verbindung zur Datenbank
  - zur Definition, Abfrage und Manipulation der Daten in der Datenbank
  - für die Zugriffskontrolle und Transaktionsverwaltung
- deklarative SQL-Anweisungen zur Deklaration von Kommunikationspuffern und SQL-Variablen
- statische SQL-Anweisungen
- dynamische SQL-Anweisungen

Das nächste Beispiel bestimmt den Vornamen des Autors White. Wenn ein Vorname gefunden wurde, wird dieser ausgegeben.

```
main()
{
  EXEC SQL BEGIN DECLARE SECTION;
  char firstName[50];
  char lastName[] = "White";
  unsigned short firstNameNull;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO server USER login.password;
  EXEC SQL SELECT auFirstName INTO :firstName :firstNameNull
    FROM Authors WHERE auLastName = :lastName;
  if (firstNameNull <> -1) printf("%s %s", firstName, lastName);
  return (0);
}
```

In Embedded SQL verwendet man Hostvariablen zur Kommunikation der Daten zwischen der Datenbank und dem Anwendungsprogramm (firstName, lastName). Innerhalb von SQL-Anweisungen wird den Hostvariablen ein „:“ vorangestellt, um sie von Datenbank-Objekten wie Tabellen oder Spalten mit demselben Namen unterscheiden zu können. Hostvariablen kann man optional Indikatorvariablen (firstNameNull) zuordnen, indem man sie in der SQL-Anweisung direkt hinter die eigentliche Hostvariable schreibt. Indikatorvariablen zeigen an, ob der Wert der Hostvariablen ein Nullwert ist oder ob zum Beispiel bei der Zuweisung Werte abgeschnitten wurden.

Im folgenden wird eine kurze Übersicht über die einzelnen Embedded SQL-Befehle gegeben.

- CONNECT TO [serverName.]databaseName [AS connectionName]
  USER login[.password]

  CONNECT baut eine Verbindung zum angegebenen SQL-Server auf und gibt ihr den angegebenen Namen.

- DISCONNECT [connectionName | CURRENT | ALL]

  DISCONNECT schließt die angegebene, die aktuelle oder alle Verbindungen.

- SET CONNECTION connectionName

  SET CONNECTION macht die angegebene Verbindung zur aktuellen Verbindung.

- DECLARE cursorName CURSOR FOR (selectStmt | preparedStmt)

  DECLARE CURSOR deklariert einen Cursor zum zeilenweisen Abfragen der Ergebnisse der SQL-Anweisung.

Beispiel: EXEC SQL DECLARE C1 CURSOR FOR
SELECT auFirstName, auLastName FROM Authors FOR BROWSE;

- OPEN cursorName [(USING DESCRIPTOR :sqlda
  | :hostVar {"," :hostVar})]

  OPEN liest den ersten Datensatz der Ergebnismenge der SQL-Anweisung, die zu dem angegebenen Cursor gehört. Für zur Laufzeit erstellte SQL-Anweisungen muß man eine SQLDA (SQL Descriptor Area) Struktur übergeben. Wenn die SQL-Anweisung des Cursors Parameter-Platzhalter („?“) enthält, muß man für jeden Platzhalter eine Hostvariable angeben.

- FETCH cursorName [(USING DESCRIPTOR :sqlda
  | INTO :hostVar {, :hostVar })]

  FETCH liest den nächsten Datensatz der Ergebnismenge der SQL-Anweisung, die zu dem angegebenen Cursor gehört.

  Beispiel:

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT auFirstName, auLastName FROM Authors FOR BROWSE;
EXEC SQL OPEN C1;

while (SQLCODE == 0)
{
   EXEC SQL FETCH C1 INTO :firstName, :lastName;
   printf("%s %s", firstName, lastName);
}
```

- SELECT [selectList] INTO :hostVar {"," :hostVar} selectOptions

  SELECT liest die angegebenen Ergebnisse der SQL-Anweisung in die Hostvariablen ein. Liefert die SQL-Anweisung mehr als eine Ergebniszeile, wird ein Fehler gemeldet.

  Beispiel: EXEC SQL SELECT auLastName INTO :name FROM Authors
  WHERE storID = :id;

- DESCRIBE preparedStmt INTO :sqlda

  DESCRIBE füllt Einträge in der Datenstruktur sqlda mit Informationen über die vorbereitete SQL-Anweisung.

Beispiel:
```
#define NumReturnCols 2
struct sqlda *mysqlda;
EXEC SQL BEGIN DECLARE SECTION;
char statement[] = "SELECT auFirstName, auLastName FROM Authors";
EXEC SQL END DECLARE SECTION;

mysqlda = malloc(SQLDASIZE(NumReturnCols));
mysqlda->sqln = NumReturnCols;
mysqlda->sqldabc = SQLDASIZE(NumReturnCols);

EXEC SQL DECLARE C1 CURSOR FOR stmt1;
EXEC SQL PREPARE stmt1 FROM :statement;
EXEC SQL DESCRIBE stmt1 INTO :mysqlda;

EXEC SQL OPEN C1;
EXEC SQL FETCH C1 USING DESCRIPTOR :mysqlda;
```

- PREPARE stmtName [INTO :sqlda] FROM :hostVar

  PREPARE bereitet die SQL-Anweisung, die als Zeichenkette in der angegebenen Hostvariable steht, zur Ausführung vor und assoziiert den angegebenen Namen mit dieser SQL-Anweisung.

  Beispiel:
```
EXEC SQL BEGIN DECLARE SECTION;
char prep[] = "insert into table values(?,?,?)";
char name[30];
char car[30];
double num;
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE prepStat FROM :prep;

while (SQLCODE == 0)
{
   strcpy(name, "Elaine");
   strcpy(car, "Lamborghini");
   num = 4.9;
   EXEC SQL EXECUTE prepStat USING :name, :car, :num;
}
```

- EXECUTE preparedStmt [(USING DESCRIPTOR :sqlda
  | :hostVar {"," :hostVar})]

EXECUTE führt die vorbereitete SQL-Anweisung aus, wobei die Hostvariable sqlda beschreibende Informationen enthält, und für jeden Parameter-Platzhalter („?“) in der SQL-Anweisung eine weitere Hostvariable angegeben werden muß.

- EXECUTE IMMEDIATE :stmt

  EXECUTE IMMEDIATE führt die SQL-Anweisung aus, die als Zeichenkette in der angegebenen Hostvariablen steht.

- DELETE [FROM] (tableName | viewName)
  WHERE (CURRENT OF cursorName | searchConditions)

  DELETE löscht den Datensatz, über dem der angegebene Cursor gerade steht, oder alle Datensätze, die die angegebene Bedingung erfüllen.

- UPDATE (tableName | viewName) SET (col "=" expr) {"," (col "=" expr) }
  WHERE CURRENT OF cursorName
  UPDATE (tableName | viewName)
  SET [tableName. | viewName.] columnName "="
  (expr | NULL | „(“selectStatement„)“)
  {, (expr | NULL | „(“selectStatement„)“)}
  [FROM (tableName | viewName) {, (tableName | viewName)}]
  [WHERE searchCondition]

  UPDATE ändert den Datensatz, über dem der angegebene Cursor gerade steht, oder alle Datensätze, die die angegebene Bedingung erfüllen.

- CLOSE cursorName

  CLOSE schließt den angegebenen Cursor, wobei nichtbearbeitete Daten verworfen werden.

- WHENEVER (NOT FOUND | SQLERROR | SQLWARNING)
  (CONTINUE | GOTO stmtLabel | CALL function())

  WHENEVER legt fest, was beim Auftreten eines Fehlers passieren soll.

1.3.5 Statisches SQL

Statisches SQL umfaßt SQL-Anweisungen, die man zur Übersetzungszeit exakt kennt (d.h. die als Zeichenketten vorgegeben sind) und die insbesondere keine Hostvariablen für Tabellen- oder Spaltennamen enthalten. Solche Anweisungen können schon zur Übersetzungszeit vorbereitet werden, datenbankinterne Berechnungen können vorgezogen werden (zum Beispiel die Erstellung des Zugriffsplanes für die Anweisung). Man kennt auch die Anzahl und Typen der Ergebnisspalten der SQL-Anweisung, die bei jedem Programmablauf in exakt gleicher Weise ausgeführt wird.

Statisches SQL ist effizient, da die Anweisungen vor ihrer Ausführung einmal vorbereitet werden und dann (mehrere Male) ausgeführt werden können. Die Anwendung, die statisches Embedded SQL verwendet, ist allerdings nach der Übersetzung an eine spezielle Datenbank gebunden.

1.3.6 Dynamisches SQL

Dynamisches SQL erlaubt die Definition und Ausführung von SQL-Anweisungen zur Laufzeit. Teile der Anweisungen befinden sich in Programmvariablen, die ihre Werte zur Laufzeit natürlich ändern können. Es sind somit komplexere und sehr flexible Programme unter Einbeziehung von Benutzereingaben zur Laufzeit möglich (der Benutzer kann zum Beispiel die Suchbedingung einer WHERE-Klausel interaktiv eingeben). Innerhalb einer SQL-Anweisung steht ein Fragezeichen für einen Platzhalter eines Variablenwerts.

Normalerweise wird die Ausführung von Befehlen in dynamischem SQL mit PREPARE vorbereitet und anschließend mit EXECUTE ausgeführt. Diese beiden Schritte können zu EXECUTE IMMEDIATE kombiniert werden.

Dynamisches SQL hat allerdings neben dem Vorteil der Flexibilität auch einige Nachteile:

- kompliziertere Programmierung
- Verwendung spezieller Datenstrukturen
- längere Laufzeit

Dynamisches SQL ist aber notwendig, wenn die Anwendung zum Zeitpunkt der Übersetzung die auszuführende SQL-Anweisung, die Anzahl und die Datentypen der Hostvariablen oder die Namen der Datenbankobjekte (Tabellen, Sichten, Spalten, Indizes, Benutzernamen) nicht kennt.

Je nachdem, was alles zur Übersetzungszeit nicht bekannt ist, was also dynamisch ist, ist die Programmierung von dynamischem SQL mehr oder weniger aufwendig.

- Für eine DML-Anweisung, die keine Abfrage ist und keine Eingabe-Hostvariablen verwendet, muß man zur Laufzeit nur die Zeichenkette für den Befehl erzeugen und an EXECUTE IMMEDIATE übergeben.

- Bei einer DML-Anweisung, die keine Abfrage ist und eine bekannte Anzahl von Eingabe-Hostvariablen mit bekannten Datentypen verwendet, muß man nach dem Erzeugen der Zeichenkette des Befehls die SQL-Anweisung mit PREPARE vorbereiten und bei der Ausführung die Werte der Eingabe-Hostvariablen an die Anweisung binden.

- Bei einer Abfrage mit einer bekannten Anzahl von Attributen und Eingabe-Hostvariablen, deren Datentypen alle bekannt sind, erzeugt man zuerst die Zeichenkette des Befehls, bereitet die SQL-Anweisung vor, führt sie aus und kann dann die Datensätze, solange welche vorhanden sind, mit FETCH auslesen.

- Für eine Abfrage mit einer unbekannten Anzahl von Attributen und Eingabe-Hostvariablen, deren Datentypen nicht bekannt sind, muß man schließlich zur Laufzeit Speicherplatz für die Datenkommunikation anlegen und insgesamt 18 Schritte beachten (siehe auch [SQL2]).

1.4 ODBC

ODBC (Open Database Connectivity) [ODBC94] ist eine von Microsoft definierte Programmierschnittstelle für den Zugriff von Programmen auf Datenbanken mittels SQL. Für den Programmierer ist ODBC eine Funktionsbibliothek, bei deren Verwendung es nicht notwendig ist, das Programm mit einem Precompiler zu übersetzen oder eine neue Sprache wie Embedded SQL zu lernen (dafür aber eine Funktionsbibliothek). Weitere Vorteile von ODBC sind die Unterstützung durch mehrere Hersteller auf mehreren Plattformen und die Transparenz bezüglich der Netzverbindung. Die Architektur von ODBC hat vier Komponenten (siehe Abbildung 1.3):

- das Programm, das die Daten bearbeitet und über ODBC-Aufrufe SQL-Anweisungen an die Datenbank schickt;
- der Treiber-Manager, der den entsprechenden Datenbanktreiber bei Bedarf nachlädt;
- der Datenbanktreiber, der die ODBC-Aufrufe bearbeitet, SQL-Befehle an das Datenbanksystem schickt und die Ergebnisse zurückgibt und
- die Datenquelle, die in der Regel ein Datenbanksystem ist.

Eine Anwendung kann somit mit denselben Befehlen Daten von beliebige Datenbanken (für die es einen ODBC-Treiber gibt) auslesen, ändern und löschen. Diese Unabhängigkeit bezeichnet man unter anderem als *Interoperabilität* (weil die Anwendung mit beliebigen Datenbanken zusammenarbeiten kann).

ODBC definiert bestimmte Konformitätsebenen (conformance levels) in Bezug auf das unterstützte ODBC API und die unterstützte SQL-Grammatik.

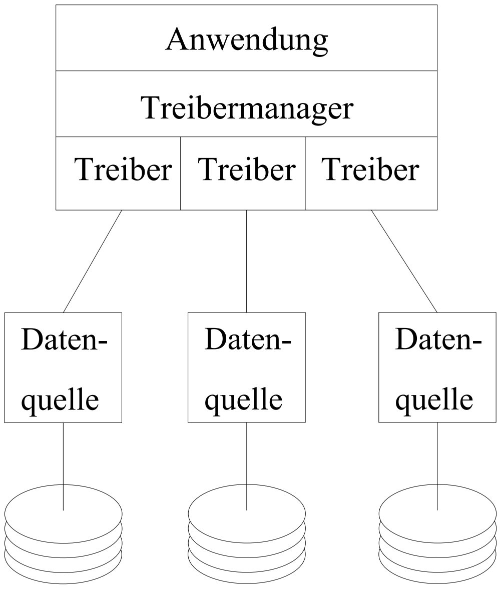

Abbildung 1.3: Architektur von ODBC

Die Grundmenge der ODBC-Funktionen entspricht den Funktionen in der X/Open und SQL Access Group Call Level Interface-Spezifikation und umfaßt:

- Anlegen und Freigeben von Hilfszeigern (handles) für das Environment, für Verbindungen und Anweisungen
- Aufbauen der Verbindungen zu Datenquellen
- Vorbereiten und Ausführen von SQL-Anweisungen, direktes Ausführen von SQL-Anweisungen
- Zuweisen von Speicherbereichen für die Datenkommunikation (Parameter und Ergebnismenge)
- Auslesen der Typinformation über die Datensätze der Ergebnismenge einer SQL-Anweisung und Auslesen der Datensätze selber
- Verwalten von Transaktionen (Commit und Rollback)
- Abfragen der Fehlerinformation

Die erste Konformitätsebene (Level 1 API) umfaßt zusätzlich:

- Aufbauen der Verbindung mit treiberspezifischen Dialogboxen
- Abfragen und Setzen von Anweisungs- und Verbindungsoptionen

- Schrittweises Senden von Parameterwerten (für lange Daten)
- Schrittweises Empfangen von Ergebnissen (für lange Daten)
- Erfragen von Kataloginformationen (über Tabellen, Spalten von Tabellen, spezielle Spalten und Statistiken)
- Abfragen von Fähigkeiten des Datentreibers und der Datenquelle wie unterstützte Datentypen, skalare Funktionen und ODBC-Funktionen

Die zweite Konformitätsebene (Level 2 API) umfaßt schließlich noch:

- Auflisten der Informationen über eine Verbindung und über verfügbare Datenquellen
- Blockweises Senden von Parameterwerten
- Abfragen der Anzahl der Parameter einer SQL-Anweisung und Erfragen der Typinformation der Parameter
- Cursor zum datensatzweisen Auslesen der Ergebnismenge
- Abfragen der datenbankspezifischen Form einer SQL-Anweisung
- Erfragen erweiterter Kataloginformation (über Privilegien, Schlüssel und Prozeduren)
- Aufrufen einer DLL[2] zur Datenkonversion

Die minimale SQL-Grammatik umfaßt:

- Befehle zur Datendefinition: CREATE TABLE und DROP TABLE
- Befehle zur Datenmanipulation: einfache Form des SELECT, INSERT, UPDATE (mit Suchbedingung) und DELETE (mit Suchbedingung)
- einfache Ausdrücke (wie zum Beispiel $A > B + C$)
- die Datentypen CHAR und VARCHAR oder LONG VARCHAR

Von den meisten Datenbanktreibern werden zusätzlich angeboten:

- weitere Befehle zur Datendefinition: ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT und REVOKE
- erweiterter Befehl zur Datenmanipulation: SELECT mit GROUP und HAVING

[2] dynamic link library, dynamisch ladbare Bibliothek

- komplexere Ausdrücke wie Unterabfragen und mengenwertige Funktionen wie SUM und MIN
- weitere Datentypen: DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT und DOUBLE PRECISION

Die vollständige SQL-Grammatik umfaßt schließlich noch:

- weitere Befehle zur Datenmanipulation: äußerer Verbund (outer join), positioniertes UPDATE, positioniertes DELETE, SELECT FOR UPDATE und Vereinigungen von Ergebnismengen
- skalare Funktionen wie SUBSTRING, ABS und Konstanten für die Datentypen Date, Time und Timestamp in Ausdrücken
- weitere Datentypen: BIT, TINYINT, BIGINT, BINARY, VARBINARY, LONG VARBINARY, DATE, TIME und TIMESTAMP
- Prozeduraufrufe

Die meisten Funktionen der ODBC-Schnittstelle lassen sich in einfacher Weise auf die Konstrukte von Embedded SQL abbilden. ODBC verwendet allerdings anstelle von Hostvariablen Parameter-Platzhalter.

In [ODBC94] findet man eine detaillierte Gegenüberstellung der Konstrukte von Embedded SQL und der Funktionen von ODBC, die einem Anwendungsprogrammierer das Umsteigen auf die jeweils andere Art erleichtert.

1.5 Entity-Relationship Modell

Im Kapitel 3 werden wir zur Modellierung der Stücklistenverwaltung das Entity-Relationship Modell verwenden. Der Begriff des Entity-Relationship Modells geht auf Chen zurück. Die Grundidee sei hier kurz skizziert: Das Ziel ist die Datenmodellierung der interessierenden Aspekte des Problems. Diese erfolgt durch Objekte und Beziehungen zwischen Objekten. Im Entity-Relationship Modell bezeichnen Entitäten Objekte und werden durch Rechtecke dargestellt. Relationships bezeichnen Beziehungen zwischen Entitäten und werden durch Kanten und Rauten dargestellt. Generalisierungen (Ist-Beziehungen) werden durch Kanten und Dreiecke dargestellt. Eine genauere Beschreibung findet man in [Chen76] und [Heu95].

Kapitel 2

Modulbeschreibungen

Im Rahmen dieser Diplomarbeit ist die Anbindung kommerziell verfügbarer Datenbanken mittels ODBC an Oberon implementiert worden. Das Ziel war ein einfacher Zugriff auf die in der Datenbank gespeicherten Daten. Die Abfragen werden dabei oft in SQL formuliert. Um die Abfragen in das Anwendungsprogramm integrieren zu können, muß man entweder die Programmiersprache um die SQL-Syntax erweitern oder Funktionen zur Verfügung haben, die den SQL-Abfragen entsprechen. Wir haben den zweiten Ansatz gewählt. Es ist nun auf einfache Weise möglich, Datenbankabfragen in SQL als Zeichenketten anzugeben, an die Datenbank zu schicken und die Ergebnisse abzufragen.

2.1 Überblick über die implementierten Module

Das Basismodul ODBC ist eine direkte Abbildung des Bibliothek ODBC32.DLL, die unter Windows den Treibermanager implementiert. Dieses Modul wird von allen weiteren Modulen importiert (siehe Abbildung 2.1).

Das Modul Ref ist ein weiteres Basismodul. Es wurde im Rahmen der Diplomarbeit entwickelt und ist jetzt Teil des Oberon-Systems[1] [WiG89]. Dieses Modul wird vom Modul EmbeddedSQL verwendet, um von Variablennamen, die in SQL-Abfragen vorkommen, auf deren Speicheradressen zu kommen.

EmbeddedSQL bietet eine einfach zu handhabende ODBC-Schnittstelle an. EmbeddedSQL definiert eine Klasse Connection für die Verbindung zu einer Datenbank und eine Klasse Statement für SQL-Abfragen. Innerhalb von SQL-Abfragen können Oberon-Variablen vorkommen, die bei der Ausführung der Abfrage dann durch die Werte der Variablen ersetzt werden bzw. die die Ergebnisse der Abfragen empfangen. Man kann bei der Definition einer Tabelle einfach den Typ eines Records angeben - die Tabelle wird dann so angelegt, daß es für jedes Feld des Records eine Spalte mit geeignetem Datentyp gibt. Auch beim Einfügen von Datensätzen und

[1]Das Oberon-System ist frei verfügbar über die Internet-Adresse http://www.ssw.uni-linz.ac.at/pub/Oberon

beim Auslesen gibt man einfach Oberon-Records an. Dieses Modul bietet aber nicht die Möglichkeit, Daten aus beliebigen Tabellen auszulesen. Man muß schließlich in den SQL-Abfragen Oberon-Variablen für die Ergebnisse angeben. Die Typen dieser Variablen müssen den Typen der Ergebnisse entsprechen, diese Typen kennt man wiederum im allgemeinen noch nicht zur Übersetzungszeit.

SQL bietet im Vergleich zu EmbeddedSQL eine etwas erweiterte Funktionalität an. Insbesondere ist ein allgemeiner Mechanismus vorgesehen für die Kommunikation von Daten beliebiger Typen. Es ist somit möglich einen Datenbankbrowser zu implementieren, bei dem der Benutzer zur Laufzeit SQL-Abfragen eingibt und die Ergebnisse angezeigt werden. Die Typen der Ergebnisse müssen nicht zur Übersetzungszeit bekannt sein. Das Modul SQL unterstützt alle ODBC-Datentypen (siehe Kapitel 2.2.1), z.B. auch lange binäre Daten (BLOBs, binary large objects). Solche dateibasierten Datentypen werden im Modul EmbeddedSQL nicht unterstützt.

Das Modul SQLExt implementiert die vollständige Funktionalität der ODBC Konformitätsebenen 1, 2 und 3 (siehe Kapitel 1.4). Wenn man mit der Funktionalität des Moduls SQL nicht auskommt, kann man auf die Erweiterung SQLExt zurückgreifen. Weiters stellt SQLExt einige Hilfsfunktionen für die Umsetzung von Datentypen auf ihre entsprechenden Zeichenketten an (z.B. Umsetzung des ODBC-Datentyps mit der Nummer −1 (LONG VARYING CHARACTER) mit 32000 Zeichen auf den MS Access-Datentyp „MEMO(32000)“).

Schlußendlich wurden noch Anwendungsprogramme implementiert: DBBrowser ist ein Datenbankbrowser zum Administrieren von Datenbanken und Abfragen von Daten aus beliebigen Datenbanken. DataEditor ist ein datensatzorientierter Editor. Man kann neue Datensätze einfügen, bestehende ändern oder löschen und nach Datensätzen suchen. ODBCBench ist ein interoperables Testprogramm, das ohne Neuübersetzung mit beliebigen Datenbanken arbeiten kann (Tabellen erstellen, Daten einfügen und Daten auslesen). Addresses schließlich ist ein Beispielprogramm, das demonstriert, wie einfach es ist, eine kleine Adressenverwaltung mit EmbeddedSQL zu implementieren.

Die Tabelle 2.1 gibt eine Aufstellung der implementierten Module mit deren Größe und Zweck. In der Abbildung 2.1 ist die entstandene Modulhierarchie dargestellt, wobei die Pfeile die Importbeziehung andeuten (importierendes Modul bei der Pfeilspitze).

2.2 ODBC

Das Modul ODBC enthält Deklarationen von Typen und Konstanten und Prozeduren, die in der von Microsoft definierten ODBC-Schnittstelle [ODBC94] enthalten sind. Im wesentlichen ist es eine 1:1-Abbildung der DLL ODBC32.DLL, die unter Windows den Treibermanager implementiert.
Das Definitionsmodul wurde so gehalten, daß es möglichst gleich dem unter Oberon/F [Pou95] verfügbaren ist, um die Portabilität von Oberon-ODBC-Programmen

Modulname	Zeilen	Objektcodegröße	Zweck
ODBC	314	1645	Abbildung der ODBC32.DLL
Ref	518	7210	Zugriff auf Metainformation
EmbeddedSQL	830	12976	einfache Schnittstelle
SQL	896	12769	flexiblere Schnittstelle
SQLExt	240	5075	Erweiterungen zu SQL
DBBrowser	735	14762	Datenbank-Browser
DataEditor	414	8819	generischer Datensatz-Editor
ODBCBench	495	6786	interoperables Testprogramm
Addresses	73	963	EmbeddedSQL Beispielprogramm

Tabelle 2.1: Implementierte Module

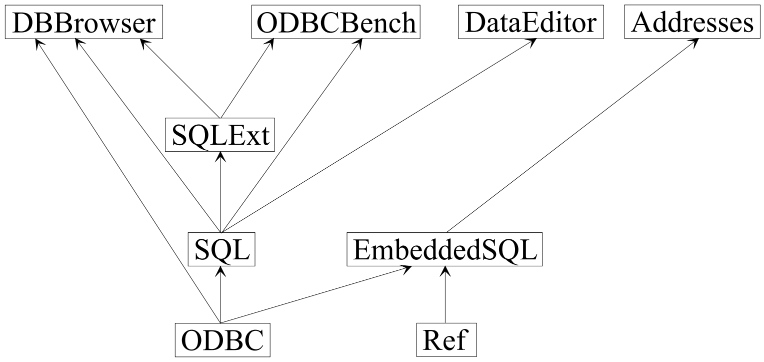

Abbildung 2.1: Modulhierarchie

nicht einzuschränken.

Im Anhang A.1 ist das komplette Definitionsmodul abgedruckt, hier soll nur ein Überblick darüber, was im Modul ODBC implementiert ist, gegeben werden.

Das Modul ODBC definiert:

- Konstanten für die vordefinierten Datentypen in ODBC, in C und in Oberon
- Konstanten für Rückgabewerte
- Konstanten für Parameter von ODBC-Funktionen
- die Datentypen Time, Date und TimeStamp
- Prozedurvariablen für die ODBC-Funktionen

Es enthält aber keine weiteren Prozeduren oder globale Variablen und hat somit selber keinen globalen Zustand, sondern ist, wie gesagt, lediglich eine direkte Abbildung der DLL. Es ist somit einfach möglich, bestehende ODBC-Anwendungsprogramme, die in anderen Sprachen implementiert sind (zum Beispiel in C), nach Oberon zu portieren.

2.2.1 Datentypen

Nicht jeder Datenbanktreiber unterstützt alle definierten SQL-Datentypen, da verschiedene Datenbanksysteme unterschiedliche Datentypen zur Verfügung stellen.

Es ist nun leider nicht so, daß ODBC eine automatische Abbildung von vordefinierten Datentypen (z.B. CHARACTER VARYING, NUMERIC, INTEGER, CHARACTER) auf die vom jeweiligen Datenbanksystem unterstützten Datentypen (z.B. MEMO, CURRENCY, LONG, TEXT) durchführt, vielmehr wird das dem Anwendungsprogrammierer überlassen. Dieser Punkt führt in der Folge dazu, daß die Interoperabilität, die man aufgrund des ODBC-Architektur hätte, wieder teilweise verloren geht, wenn das Anwendungsprogramm nicht selber darauf achtet, mit beliebigen Datenbanken (und somit Datentypen) arbeiten zu können [St96]. Das mag in vielen Fällen gar keine große Einschränkung sein, da Anwendungen immer noch nicht ganz von der Datenbank unabhängig sind (trotz der oft angesprochen logischen und physischen Datenunabhängigkeit). Dennoch wurde in der vorliegenden Diplomarbeit versucht, den Anwendungsprogrammierer auf dem Weg zu interoperablen Oberon-Programmen zu unterstützen.

Die folgende Tabelle gibt einen Einblick in die Unterschiedlichkeit der unterstützten Datentypen einiger Datenbanktreiber. Alle Felder, die leer gelassen wurden, zeigen vom jeweiligen Datenbanktreiber nicht unterstützte Datentypen an. Die Breite eines Datentyps ist die maximale Anzahl von Stellen (Mantissenlänge, Anzahl dezimaler Ziffern, Anzahl von Zeichen bei Konvertierung in Zeichenkette).

Der Programmmierer darf also keine Annahmen über verfügbare Datentypen machen. Er muß sogar bei SQL-Anweisungen, die Datentypen enthalten (also zum Beispiel CREATE TABLE) genau die Zeichenkette für den Datentyp angeben, die der Datenbanktreiber versteht.

Diese Last kann man dem Anwendungsprogrammierer natürlich nicht aufbürden, weshalb dafür im anschließend beschriebenen Modul SQL einige Prozeduren zum Umsetzen der gewünschten Datentypen auf die unterstützten Datentypen zur Verfügung gestellt werden.

Was ODBC allerdings schon anbietet, ist Datenkonversion. Diese findet statt, wenn man Daten vom Anwendungsprogramm an die Datenbank schickt oder von der Datenbank Daten erhält. Man muß allerdings als Anwendungsprogrammierer diese Datenkonversion wiederum explizit veranlassen.

SQL Datentyp	Oracle	Breite	Access	Breite	Excel	Breite	CSV	Breite
BIT			BIT	1	LOGICAL	1		
TINYINT			UNSIGNEDBYTE	3				
BIGINT								
LONG VARBINARY	LONG RAW	2147483647	LONGBINARY	128000000				
VARBINARY	RAW	255						
BINARY								
LONG VARCHAR	LONG	2147483647	MEMO	128000000			LONGCHAR	65500
CHAR	CHAR	255	TEXT	255			CHAR	1
NUMERIC	NUMBER	38	CURRENCY	18				
DECIMAL								
INTEGER			LONG	10			INTEGER	10
SMALLINT			SHORT	5				
FLOAT	FLOAT	15						
REAL			SINGLEFLOAT	7				
DOUBLE PRECISION			DOUBLEFLOAT	15	NUMBER	15	FLOAT	15
DATE					DATE	10	DATE	10
TIME					TIME	8		
TIMESTAMP	DATE	19	DATETIME	19	DATETIME	19		
VARCHAR	VARCHAR2	2000			TEXT	255		

2.2.2 Ausführung von ODBC-Anweisungen

Im folgenden wird der Ablauf typischer Anwendungsfälle skizziert. Es werden dabei schon die Namen der von ODBC definierten Prozeduren verwendet. Eine Übersicht über alle in ODBC definierten Operationen bietet das Definitionsmodul ODBC im Anhang A.1.

- Im allgemeinen Fall muß man die Abarbeitung einer SQL-Anweisung (Zeichenkette) zuerst mit SQLPrepareStatement vorbereiten.

- Anschließend muß man, falls die Anweisung Parameter hat, Variablen zur Datenkommunikation an die Anweisung mit SQLBindParameters binden.

- Nach dem Vorbereiten und Binden der Parameter führt man die Anweisung mit SQLExecute aus.
- Die weitere Abarbeitung hängt von der Art der Anweisung ab, ob man sie wiederholt ausführen möchte (zum Beispiel ein INSERT mit geänderten Parameterwerten) oder nicht. Wenn man die Anweisung also erneut ausführen will, ruft man SQLExecute weiter auf.
- Handelt es sich bei der Anweisung um ein SELECT, werden also Daten von der Datenbank geliefert, muß man für die Datenkommunikation Variablen mit SQLBindCol an die einzelnen Spalten der Ergebnismenge binden und kann nach dem Ausführen von SQLExecute mit SQLFetch die Datensätze der Reihe nach auslesen.
- Handelt es sich bei der Anweisung um ein UPDATE, DELETE oder INSERT, kann man nach der Ausführung mit SQLRowCount die Anzahl der betroffenen Datensätze erfragen.

2.3 EmbeddedSQL

EmbeddedSQL bietet eine einfach zu handhabende ODBC-Schnittstelle an. Unterstützt wird der vollständige SQL-Umfang (wie vom Datenbanktreiber implementiert) sowie die automatische Umsetzung von Datentypen (siehe Kapitel 2.2.1).

Zum Datentransfer zwischen der Datenbank und dem Programm werden normale Oberonvarablen innerhalb von SQL-Anweisungen verwendet. Um diese Variablen von Namen, die in der Datenbank verwendet werden (zum Beispiel die Namen von Tabellen und Spalten), zu unterscheiden, wird ihnen ein Doppelpunkt vorangestellt.

In der SQL-Anweisung „SELECT firstName FROM Persons WHERE age > :minAge INTO :name“ sind minAge und name Oberonvariablen. minAge ist eine

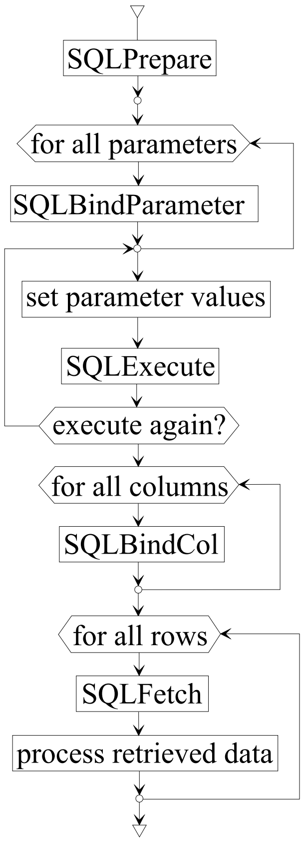

Abbildung 2.2: Ausführung von Anweisungen

Eingangsvariable, name ist eine Ausgangsvariable. Variablen können entweder skalar sein oder Records. Wenn man eine Rekordvariable angibt, wird diese implizit zur Liste der Komponenten expandiert. Die Anweisung „SELECT * FROM Persons INTO :person" ist also äquivalent zur Anweisung „SELECT * FROM Persons INTO :person.firstName, :person.lastName, :person.age".

Bei der Umsetzung der SQL-Anweisungen, die Oberonvariablen enthalten, in SQL-Anweisungen, die der ODBC-Treiber bearbeiten kann, müssen die Bezeichner der Oberonvariablen durch „?" ersetzt werden. Zusätzlich müssen dem ODBC-Treiber die Puffer zur Datenkommunikation der Eingangs- und Ausgangsvariablen bekanntgegeben werden. Um von den Variablennamen auf die Adressen und Typen der Variablen zu kommen, wird das Modul Ref [StM96] verwendet, das Zugriff auf die Referenzinformation von geladenen Oberon Modulen bietet. Die Referenzinformation entspricht der Symbolliste des Compilers und enthält unter anderem folgende Informationen: Typen, Prozeduren und globale Variablen eines Moduls, Parameter und lokale Variablen von Prozeduren, Felder von Records.

```
DEFINITION EmbeddedSQL;

CONST
  InvHandle = -2; Error = -1; Success = 0; (* return codes *)
  SuccessWithInfo = 1; NoDataFound = 100;
  showErrors = 1; haltOnErrors = 2; verboseErrorMsg = 3; (* options *)

TYPE
  Connection = POINTER TO ConnectionD;
  ConnectionD = RECORD
    ret: INTEGER; (* return code of the last operation *)
    dbc-: LONGINT; (* connection handle *)
    PROCEDURE (c: Connection) Prepare (sqlString: ARRAY OF CHAR): Statement;
    PROCEDURE (c: Connection) Error (str: ARRAY OF CHAR);
    PROCEDURE (c: Connection) Commit;
    PROCEDURE (c: Connection) Rollback;
    PROCEDURE (c: Connection) GetIntInfo (infoNr: INTEGER; VAR info: LONGINT);
    PROCEDURE (c: Connection) GetStrInfo (infoNr: INTEGER;
      VAR info: ARRAY OF CHAR);
    PROCEDURE (c: Connection) GetIntOption (option: INTEGER;
      VAR info: LONGINT);
    PROCEDURE (c: Connection) SetIntOption (option: INTEGER; info: LONGINT);
    PROCEDURE (c: Connection) GetStrOption (option: INTEGER;
      VAR info: ARRAY OF CHAR);
    PROCEDURE (c: Connection) SetStrOption (option: INTEGER;
      info: ARRAY OF CHAR);
  END ;

  Statement = POINTER TO StatementD;
  StatementD = RECORD
    ret: INTEGER; (* return code of the last operation *)
    stmt-: LONGINT; (* statement handle *)
    conn-: Connection; (* the connection on which the statement is executed *)
    PROCEDURE (s: Statement) Execute;
    PROCEDURE (s: Statement) IsNull (varName: ARRAY OF CHAR): BOOLEAN;
    PROCEDURE (s: Statement) SetNull (varName: ARRAY OF CHAR);
    PROCEDURE (s: Statement) Fetch (): BOOLEAN;
    PROCEDURE (s: Statement) Error (str: ARRAY OF CHAR);
    PROCEDURE (s: Statement) RowCount (): LONGINT;
    PROCEDURE (s: Statement) GetOption (option: INTEGER; VAR info: LONGINT);
    PROCEDURE (s: Statement) SetOption (option: INTEGER; info: LONGINT);
  END ;

VAR
  env-: ODBC.ENV; (* environment handle for the ODBC session *)
  options: SET;
```

```
PROCEDURE Open (source, user, passwd: ARRAY OF CHAR): Connection;
PROCEDURE OpenUI (connStrIn: ARRAY OF CHAR;
  VAR connStrOut: ARRAY OF CHAR): Connection;
PROCEDURE SetOption;
PROCEDURE SQLError (env, dbc, stmt: LONGINT; VAR sqlState: ARRAY OF CHAR;
  VAR nativeErr: LONGINT; VAR errMsg: ARRAY OF CHAR): INTEGER;
PROCEDURE ShowError (ret: INTEGER; str: ARRAY OF CHAR; env, dbc,
  stmt: LONGINT);

END EmbeddedSQL.
```

Globaler Zustand

- env ist ein Verweis auf einen Speicherbereich, den der ODBC-Treibermanager zur Durchführung der Datenbankoperationen benötigt. Dieser Verweis ist gültig, solange das Modul geladen ist.

- options ist eine Menge von Optionen, die entscheiden, wie auf Fehler reagiert werden soll. Wenn showErrors in der Menge enthalten ist, werden Fehler weiter behandelt, d.h. es wird Information über den Fehler auf die Standardausgabe geschrieben. Ist verboseErrorMsg gesetzt, erfolgt noch etwas mehr Ausgabe, bei haltOnErrors wird die Ausführung abgebrochen. Standardmäßig ist showErrors gesetzt.

Typen

- Connection stellt einen gültigen Kommunikationskanal zwischen dem Programm und der Datenbank dar. Über diesen werden Befehle übertragen und Ergebnisse zurückgeliefert. ret zeigt den Erfolg der letzten Operation an, dbc ist ein Verweis auf einen Speicherbereich des ODBC-Treibers zur Verwaltung der Verbindung.

- Statement stellt eine SQL-Anweisung dar, die zur Ausführung über der Verbindung conn vorbereitet wurde. ret zeigt den Erfolg der letzten Operation an, stmt ist ein Verweis auf einen Speicherbereich des ODBC-Treibers zur Verwaltung der SQL-Anweisung.

2.3.1 Operationen

- conn := Open(source, user, password) öffnet eine Verbindung zur Datenbank source mit der Benutzerkennung user und dem Paßwort password. Verbindungen werden automatisch geschlossen, wenn sie nicht mehr verwendet werden.

- conn := OpenUI(connStrIn, connStrOut) öffnet ebenfalls eine Verbindung zur Datenbank, allerdings ist es möglich, die Eingaben interaktiv mit Dialogen des Datenbanktreibers durchzuführen. Die Zeichenkette connStrOut enthält nach dem Aufruf alle eingegebenen Informationen und kann bei späteren Verbindungsaufbaus wiederverwendet werden.

- SetOption ist ein Kommando, mit dem man die Optionen zur Fehlerbehandlung setzen kann. Indem man die negativen Werte der Optionen angibt, setzt man sie zurück.

- Mit err := SQLError (env, dbc, stmt, sqlState nativeErr, errMsg) kann man zusätzliche Informationen über den letzten Fehler erfragen, der bei der Abarbeitung der SQL-Anweisung stmt auf der Verbindung dbc des ODBC-Treibermanagers env aufgetreten ist. err ist der Fehlerkode, sqlState ist eine kurze Zeichenkette, die den Zustand des Datenbanktreibers wiedergibt, nativeErr ist ein datentreiberspezifischer Fehlerkode, und errMsg enthält die Fehlermeldung in textueller Form.

- ShowError (ret, str, env, dbc, stmt) kann man zur Ausgabe des Fehlerkodes ret und der Fehlermeldung str (wieder mit env, dbc und stmt) verwenden. Die Aufrufe stat.Error(str) und conn.Error(str) werden auf ShowError abgebildet (mit entsprechenden Werten für die anderen Parameter).

- stat := conn.Prepare(s) bereitet eine SQL-Anweisung s (als Zeichenkette angegeben) für die Ausführung vor.

- conn.Error(s) kann man zur Behandlung von Fehlern aufrufen. Wenn conn.ret nicht gleich Success, werden die Zeichenkette s und weitere Fehlerinformation ausgegeben (vorausgesetzt, die Option showErrors ist gesetzt).

- Die Transaktionskontrolle erfolgt mit conn.Commit und conn.Rollback. Commit macht alle Operationen seit Verbindungsaufbau oder dem letzten Commit persistent, Rollback macht sie ungeschehen.

- Mit GetIntInfo(infoNr, info) und GetStrInfo(infoNr, info) kann man generelle Informationen über den Datenbanktreiber und die Datenbank erfragen (maximale Anzahl aktiver Verbindungen, Name und Version der Datenbank,...). Gültige Werte für infoNr sind im Anhang C.3 zu finden. Abhängig von der Art der Information muß man GetIntInfo(infoNr, info) (wenn die Information als Ganzzahl oder Bitmaske zurückgeliefert wird) oder GetStrInfo(infoNr, info) (wenn die Information als Zeichenkette zurückgeliefert wird) aufrufen.

- Mit GetIntOption(option, info), SetIntOption(option, info), GetStrOption(option, info) und SetStrOption(option, info) kann man die aktuellen Werte von Optionen, die die Verbindung betreffen, auslesen und ändern. Beispiele für solche Optionen sind der Zugriffsmodus (nur lesend oder lesend und schreibend), der

Timeout beim Verbindungsaufbau und die Art und Weise der Isolation mehrerer Transaktionen. Gültige Werte für option sind im Anhang C.2 zu finden. Abhängig von der Art der Option muß man wieder die Zeichenketten- oder Ganzzahlvarianten aufrufen.

- stat.Execute führt die vorbereitete SQL-Anweisung stat aus.
- stat.Fetch überträgt die Ergebnisse der letzten Abfrage in die bei der Vorbereitung der Anweisung angegebenen Variablen. Wenn Fetch FALSE liefert, sind keine weiteren Daten vorhanden.
- stat.IsNull("name") liefert TRUE, wenn die Variable name einen Nullwert enthält.
- stat.SetNull("name") markiert den Inhalt der Variablen name als Nullwert.
- stat.Error("...") kann man zur Behandlung von Fehlern aufrufen. Wenn stat.ret nicht gleich Success, werden die angegebene Zeichenkette und weitere Fehlerinformation ausgegeben (vorausgesetzt, die Option showErrors ist gesetzt).
- cnt := stat.RowCount() liefert die Anzahl der von der Ausführung der SQL-Anweisung (UPDATE oder DELETE, nicht aber bei SELECT) betroffenen Tupel.
- Mit GetOption und SetOption kann man die aktuellen Werte von Optionen, die eine SQL-Anweisung betreffen, auslesen und ändern. Gültige Werte für option sind im Anhang C.1 zu finden. Abhängig von der Art der Option muß man die Zeichenketten- oder Ganzzahlvariante aufrufen.

2.3.2 Beispiele

In den folgenden Beispielen wollen wir eine kleine Adreßdatenbank aufbauen (das vollständige Programm ist im Anhang B.1.1 zu finden). Dafür definieren wir uns den Datentyp Address.

```
TYPE
  Address = RECORD
    firstName, lastName, street: ARRAY 32 OF CHAR;
    town: ARRAY 20 OF CHAR;
    zipCode: INTEGER
  END ;

VAR
  conn: EmbeddedSQL.Connection;
  stat: EmbeddedSQL.Statement;
```

```
BEGIN
  conn := EmbeddedSQL.Open(source, user, password);
  stat := conn.Prepare("CREATE TABLE Addresses FOR Address");
  stat.Execute();
END
```

Nach dem Öffnen der Verbindung erstellen wir eine Tabelle für die Adressen, die wir später einfügen wollen. Die Tabelle besteht aus genau sovielen Spalten, wie ein Record vom Typ Address Felder hat (mit entsprechenden Typen). Der Rekordtyp kann mit dem Modulnamen qualifiziert sein, in dem der Rekordtyp deklariert ist.

Um in die Tabelle Daten einzufügen, bereiten wir eine INSERT-Anweisung vor, in der wir die Variablen angeben, die die einzufügenden Werte enthalten. Diesen Variablen ist ein Doppelpunkt vorangestellt, um sie von Bezeichnern, die innerhalb der Datenbank verwendet werden, zu unterscheiden. Danach weisen wir den Variablen entweder Werte zu oder lassen sie Nullwerte enthalten. Schlußendlich führen wir die Anweisung aus, wobei die Werte der Variablen in die Datenbank transferiert werden. Eine Anweisung kann mit verschiedenen Variablenwerten mehrmals ausgeführt werden.

```
PROCEDURE Insert;
  VAR adr: Address;
BEGIN
  In.Open;
  stat := conn.Prepare("INSERT INTO Addresses values (:adr)");
  REPEAT
    In.String(adr.firstName); In.String(adr.lastName); In.String(adr.street);
    In.String(adr.town); In.Int(adr.zipCode);
    IF adr.street = "NULL" THEN stat.SetNull("adr.street") END ;
    IF In.Done THEN stat.Execute() END
  UNTIL ~In.Done
END Insert;
```

Um eine Liste aller Adressen einer Person zu erhalten, kann man die folgende SELECT-Anweisung ausführen. Der Eingabevariablen (in diesem Fall name) weisen wir vor der Ausführung den gewünschten Wert zu und führen die SQL-Anweisung nach der Vorbereitung aus. Weil die Tabelle Addresses für Records vom Typ Address erstellt wurde, hat eine Ergebniszeile genau in einem solchen Record Platz. Wenn uns nur die Vor- und Nachnamen interessieren, könnten wir eine SELECT-Anweisung der Art „SELECT firstName, lastName FROM Persons WHERE age >= :minAge INTO :person.firstName, :person.lastName“ ausführen.

```
PROCEDURE ListByName*;
  VAR adr: Address; name: ARRAY 32 OF CHAR;
BEGIN
  In.Open; In.String(name);
```

```
  IF In.Done THEN
    stat := conn.Prepare("SELECT * FROM Addresses WHERE lastName = :name
      INTO :adr");
    stat.Execute();
    WHILE stat.Fetch() DO
      Out.String(adr.firstName); Out.Char(" "); Out.String(adr.lastName); Out.String(", ");
      IF ~stat.IsNull("adr.street") THEN Out.String(adr.street) END ; Out.String(", ");
      Out.Int(adr.zipCode, 0); Out.Char(" "); Out.String(adr.town); Out.Ln
    END
  END
END ListByName;
```

Implementierung

Bei der Vorbereitung von SQL-Anweisungen mit Prepare werden die Zeichenketten analysiert. Vorkommende Oberonvariablen werden durch Fragezeichen ersetzt (wie vom ODBC-Datenbanktreiber erwartet) und die Adressen der Variablen werden an die Parameterplatzhalter gebunden. Werden Recordvariablen angetroffen, so werden diese expandiert (d.h. durch ihre Felder ersetzt, jedoch nur einstufig - Records in Records werden nicht expandiert). Die Typen und Adressen der Variablen liefert das Modul Ref, indem im lokalen Bereich der Prozedur, die die Vorbereitung der SQL-Anweisung angefordert hat, nach den Variablen gesucht wird. Die Variablen müssen deshalb entweder lokale Variablen oder Parameter der aufrufenden Prozedur sein. Die Syntax der CREATE TABLE-Anweisung wurde um `FOR recordType` erweitert, die Syntax der SELECT-Anweisung um `INTO oberonVar {, oberonVar}`.

2.4 SQL

Das Modul SQL bietet im Vergleich zu EmbeddedSQL eine etwas erweiterte Funktionalität an. Es sind eigene Datentypen für die Datenkommunikation zwischen dem Programm und der Datenbank definiert, die es zum Beispiel auch erlauben, beliebig lange binäre Daten zu übertragen (BLOBs, binary large objects). Solche dateibasierten Datentypen werden im Modul EmbeddedSQL nicht unterstützt.

```
DEFINITION SQL;

IMPORT Files;

CONST
  InvHandle = -2; Error = -1; Success = 0; (* return codes *)
  SuccessWithInfo = 1; NeedData = 99; NoDataFound = 100;

  showErrors = 1; haltOnErrors = 2; verboseErrorMsg = 3; (* options *)
```

```
  InParam = 1; InOutParam = 2; OutParam = 4; (* parameter types *)

  NTS = -3; NullData = -1; (* parameter values *)

  OberonChar = 1; OberonShortInt = -6; OberonBoolean = -7;
  OberonInteger = 5; OberonLongInt = 4; OberonReal = 7;
  OberonLongReal = 8; OberonBinary = -2; OberonFile = -4;
  OberonDate = 9; OberonTime = 10; OberonTimeStamp = 11;
  SQLBit = -7; SQLTinyInt = -6; SQLBigInt = -5;
  SQLLVarBin = -4; SQLVarBin = -3; SQLBin = -2;
  SQLLVarChar = -1; SQLChar = 1; SQLNumeric = 2;
  SQLDecimal = 3; SQLInteger = 4; SQLSmallInt = 5;
  SQLFloat = 6; SQLReal = 7; SQLDouble = 8;
  SQLDate = 9; SQLTime = 10; SQLTimeStamp = 11;
  SQLVarChar = 12;

TYPE
  Type = RECORD
    sqlType: INTEGER;
    prec: LONGINT; (* maximum number of significant digits *)
    scale: INTEGER (* -"- right to period *)
  END ;
  Field = POINTER TO FieldDesc;
  FieldDesc = RECORD
    name: ARRAY 32 OF CHAR;
    len: LONGINT;
    type: Type;
    isNull, nullable: BOOLEAN
  END ;
  IntField = POINTER TO IntFieldDesc;
  IntFieldDesc = RECORD (FieldDesc) i: LONGINT END ;
  FileField = POINTER TO FileFieldDesc;
  FileFieldDesc = RECORD (FieldDesc) f: Files.File END ;
  StringField = POINTER TO StringFieldDesc;
  StringFieldDesc = RECORD (FieldDesc) str: ARRAY 256 OF CHAR END ;
  BinaryField = POINTER TO BinaryFieldDesc;
  BinaryFieldDesc = RECORD (FieldDesc) b: ARRAY 256 OF CHAR END ;
  BooleanField = POINTER TO BooleanFieldDesc;
  BooleanFieldDesc = RECORD (FieldDesc) b: BOOLEAN END ;
  DateField = POINTER TO DateFieldDesc;
  DateFieldDesc = RECORD (FieldDesc) year, month, day: INTEGER END ;
  TimeField = POINTER TO TimeFieldDesc;
  TimeFieldDesc = RECORD (FieldDesc) hour, minute, second: INTEGER END ;
  TimeStampField = POINTER TO TimeStampFieldDesc;
  TimeStampFieldDesc = RECORD (FieldDesc)
    year, month, day, hour, minute, second: INTEGER;
```

```
    fraction: LONGINT
  END ;
  RealField = POINTER TO RealFieldDesc;
  RealFieldDesc = RECORD (FieldDesc) r: LONGREAL END ;
  ParamDesc = RECORD
    type, mapType: Type;
    inOut: INTEGER;
    name: ARRAY 32 OF CHAR
  END ;
  Row = POINTER TO RowDesc;
  RowDesc = RECORD cols: INTEGER END ;
  Connection = POINTER TO ConnectionD;
  ConnectionD = RECORD
    ret: INTEGER; (* error code of last operation *)
    dbc-: LONGINT (* ODBC connection handle *)
  END ;
  Statement = POINTER TO StatementD;
  StatementD = RECORD
    ret: INTEGER; (* error code of last operation *)
    stmt-: LONGINT; (* ODBC statement handle *)
    conn-: Connection; (* associated connection *)
    results, params: Row (* row of results and parameters *)
  END ;

  EnumDataSourcesCB = PROCEDURE (dataSource, description: ARRAY OF CHAR;
    VAR continue: BOOLEAN);
  EnumDataTypesCB = PROCEDURE (typeName: ARRAY OF CHAR;
    dataType: INTEGER; prec: LONGINT; literalPrefix, literalSuffix,
    createPars: ARRAY OF CHAR; nullable: INTEGER; caseSensitive: BOOLEAN;
    searchable, unsigned: INTEGER; money: BOOLEAN; autoInc: INTEGER;
    localTypeName: ARRAY OF CHAR; minScale, maxScale: INTEGER;
    VAR continue: BOOLEAN);
  EnumDriversCB = PROCEDURE (driver, attributes: ARRAY OF CHAR;
    VAR continue: BOOLEAN);

VAR
  env-: LONGINT;
  nullString-: ARRAY 2 OF CHAR;
  options: SET;

PROCEDURE Open (source, user, passwd: ARRAY OF CHAR): Connection;
PROCEDURE OpenUI (connStrIn: ARRAY OF CHAR;
  VAR connStrOut: ARRAY OF CHAR): Connection;
PROCEDURE ShowError (ret: INTEGER; str: ARRAY OF CHAR; env, dbc,
  stmt: LONGINT);
PROCEDURE SetOption;
```

```
PROCEDURE PrepareStatement (c: Connection;
  sqlString: ARRAY OF CHAR): Statement;
PROCEDURE ErrC (str: ARRAY OF CHAR; c: Connection);
PROCEDURE Tables (c: Connection): Statement;
PROCEDURE Commit (c: Connection);
PROCEDURE Rollback (c: Connection);
PROCEDURE Map (c: Connection; type: Type; VAR mapType: Type): BOOLEAN;
PROCEDURE GetTypeName (c: Connection; sqlType: INTEGER; VAR typeName,
  createParams: ARRAY OF CHAR);
PROCEDURE GetConnOption (c: Connection; option: INTEGER; info: Field);
PROCEDURE SetConnOption (c: Connection; option: INTEGER; info: Field);
PROCEDURE GetInfo (c: Connection; infoNr: INTEGER; info: Field);
PROCEDURE NewConnection (): Connection;

PROCEDURE BindParameters (s: Statement; types: ARRAY OF ParamDesc;
  numParams: INTEGER);

PROCEDURE Execute (s: Statement);
PROCEDURE RowCount (s: Statement; VAR rows: LONGINT);
PROCEDURE Fetch (s: Statement);
PROCEDURE ErrS (str: ARRAY OF CHAR; s: Statement);
PROCEDURE GetStatementOption (s: Statement; option: INTEGER; info: Field);
PROCEDURE SetStatementOption (s: Statement; option: INTEGER; info: Field);
PROCEDURE NewStatement (c: Connection): Statement;
PROCEDURE GenerateResultRow (s: Statement);

PROCEDURE FirstField (r: Row; VAR f: Field);
PROCEDURE NextField (VAR f: Field);
PROCEDURE PrevField (VAR f: Field);
PROCEDURE FindField (r: Row; name: ARRAY OF CHAR; VAR f: Field);

PROCEDURE EnumDataSources (cb: EnumDataSourcesCB);
PROCEDURE EnumDataTypes (c: Connection; cb: EnumDataTypesCB;
  sqlType: INTEGER);
PROCEDURE EnumDrivers (cb: EnumDriversCB);
PROCEDURE SQLError (env, dbc, stmt: LONGINT; VAR sqlState: ARRAY OF CHAR;
  VAR nativeErr: LONGINT; VAR errMsg: ARRAY OF CHAR): INTEGER;

END SQL.
```

Um mit einer Datenbank arbeiten zu können, muß zuerst eine Verbindung zur Datenbank hergestellt werden. Die Datenstruktur Connection stellt eine solche Verbindung dar. Nach dem Aufbau der Verbindung kann man SQL-Anweisungen ausführen. Man kann natürlich mit mehreren Verbindungen gleichzeitig arbeiten, sie werden automatisch geschlossen, wenn sie nicht mehr benötigt werden.

Allgemeine Operationen

- ShowError (ret, str, env, dbc, stmt) gibt die Fehlermeldung str und weitere Fehlerinformationen aus, wenn ret nicht gleich Success ist und showErrors eine der in options gesetzen Optionen ist. Ist auch die Option verboseErrorMsg gesetzt, erfolgen zusätzliche Ausgaben.

- Mit dem Kommando SetOption kann man Optionen setzen und rücksetzen. SetOption 1 -2 -3 setzt die Option showErrors und setzt die Optionen haltOnErrors und verboseErrorMsg zurück. Standardmäßig ist nur showErrors gesetzt.

- EnumDataSources(callback) ruft callback für jede Datenquelle auf, die über den ODBC-Treibermanager verfügbar ist. Indem man den Parameter continue der Callback-Prozedur auf FALSE setzt, kann man den Enumerierungsprozeß vorzeitig abbrechen. Das folgende Codestück gibt alle verfügbaren Datenquellen aus.

```
PROCEDURE PrintDataSources (dataSource, description: ARRAY OF CHAR;
  VAR continue: BOOLEAN);
BEGIN
  Out.String(dataSource); Out.Ln
END PrintDataSources;

BEGIN
  ...
  SQLEnumDataSources(PrintDataSources)
  ...
END
```

- EnumDrivers(callback) ruft callback für jeden über den ODBC-Treibermanager verfügbaren Datenbanktreiber auf. Indem man den Parameter continue der Callback-Prozedur auf FALSE setzt, kann man den Enumerierungsprozeß vorzeitig abbrechen.

- EnumDataTypes(conn, callback, sqlType) ruft callback für jeden Datentyp auf, der den gewünschten Datentyp sqlType implementiert und von der Datenbank, mit der man über conn verbunden ist, unterstützt wird. Es ist möglich, daß ein vordefinierter SQL-Datentyp von keinem, einem oder mehreren Datentypen der Datenbank implementiert wird. Wenn man für sqlType 0 angibt, wird über alle Datentypen der Datenbank iteriert. Weitere gültige Werte für sqlType sind SQLBit bis SQLLVarChar. Indem man den Parameter continue der Callback-Prozedur auf FALSE setzt, kann man den Enumerierungsprozeß vorzeitig abbrechen. Die Tabelle 2.2 enthält eine kurze Beschreibung der Parameter der Callback-Prozedur callback.

Parameter	Beschreibung / Werte
typeName	datenquellenabhängiger Name des Typs
dataType	Nummer des SQL-Datentyps
prec	maximale Genauigkeit
literalPrefix	Präfix bei Konstanten dieses Datentyps
literalSuffix	Suffix
createPars	Parameter bei der Datentypdefinition
nullable	Nullwerte zulässig
caseSensitive	Groß- und Kleinschreibung in Zeichenketten
searchable	in WHERE-Klauseln verwendbar, man kann danach suchen
unsigned	mit Vorzeichen
money	für Geldbeträge geeignet
autoInc	beim Einfügen automatisch auf eindeutigen Wert gesetzt
localTypeName	länderspezifischer Name des Datentyps
minScale	minimale Anzahl von Nachkommastellen
maxScale	maximale Anzahl von Nachkommastellen

Tabelle 2.2: Parameter von EnumDataTypesCB

2.4.1 Datenbankverbindung

- conn := Open(source, user, password) baut eine Verbindung zur Datenquelle source mit der Benutzerkennung user und dem Paßword password auf. Wenn man diese Angaben nicht von vorneherein kennt, ruft man conn := OpenUI(connStr, connStrOut) auf und setzt den Eingabeparameter connStr vor dem Aufruf auf die leere Zeichenkette (""). Der ODBC-Treibermanager erfragt dann über Dialogboxen interaktiv die zu verwendende Datenbank, den Benutzernamen und das Paßwort. Der Ausgabeparameter connStrOut liefert alle Benutzereingaben als Zeichenkette zurück, die man bei einem erneuten Verbindungsaufbau wieder angeben kann. Der Rückgabewert der Funktionen Open und OpenUI ist ein Zeiger auf eine Verbindungsstruktur.

- ErrC(str, conn) ruft ShowError mit entsprechenden Parameterwerten auf.

- Commit(conn) und Rollback(conn) sind die Aufrufe für die Transaktionskontrolle, sofern die entsprechende Datenbank diese unterstützt. Commit macht die Operationen seit dem Verbindungsaufbau oder seit dem letzten Commit persistent, während Rollback diese ungeschehen macht.

- mapped := Map(conn, type, mapType) versucht, den gewünschten Typ type auf einen Datentyp abzubilden, der von der über conn verbundenen Datenbank unterstützt wird. Wenn mapped TRUE ist, ist das gelungen und mapType ist der entsprechende Datentyp.

- GetTypeName(conn, sqlType, typeName, createParams) liefert den Namen des Datentyps sqlType in typeName. Wenn createParams ungleich der leeren Zeichenkette ist, muß man bei der Definition des Spaltentyps einer Tabelle entsprechende Parameter angeben (z.B. NUMERIC(10, 2), CHAR(32)). createParams enthält eine durch Kommata getrennte Liste von Parametern.

- GetConnOption(conn, option, info) liefert den aktuellen Wert der die Verbindung conn betreffenden Option option. Beispiele für solche Optionen sind der Zugriffsmodus (nur lesend oder lesend und schreibend), der Timeout beim Verbindungsaufbau und die Art und Weise der Isolation mehrerer Transaktionen. Gültige Werte für option sind im Anhang C.2 zu finden. Abhängig von der Art der Option muß man ein StringField (wenn die Information als Zeichenkette zurückgeliefert wird) oder ein IntField (wenn die Information als Ganzzahl oder Bitmaske zurückgeliefert wird) zum Auslesen der Werte übergeben.

- SetConnOption(conn, option, info) setzt die Option info auf den Wert von info.

- GetInfo(conn, infoNr, info) liefert Informationen allgemeiner Art über den Treiber und die Datenbank, die mit der Verbindung conn assoziiert sind. Man kann zum Beispiel Information über den ODBC Treiber (wie die maximale Anzahl gültiger Verbindungen, den Datenquellennamen etc.), über die Datenbank selber (Name der Datenbank, des Datenbankprogrammes, Version), über die Datenquelle (Transaktionskontrolle), über Umfang der unterstützten SQL-Grammatik, die Limits von SQL, über unterstützte skalare Funktionen und über die Datentypkonversion erfragen. Gültige Werte für infoNr sind im Anhang C.3 zu finden. Abhängig von der Art der Information muß man ein StringField (wenn sie als Zeichenkette zurückgeliefert wird) oder ein IntField (wenn sie als Ganzzahl oder Bitmaske zurückgeliefert wird) zum Auslesen der Werte übergeben.

- conn := NewConnection() liefert eine initialisierte Verbindungsstruktur, die aber noch mit keiner Datenbank verbunden ist. Diese Funktion braucht man nur für spezielle Erweiterungen, bei denen man den Verbindungsaufbau selber implementieren möchte.

2.4.2 SQL-Anweisungen

- stat := PrepareStatement(conn, sqlString) bereitet die als Zeichenkette angegebene SQL-Anweisung sqlString zur Ausführung über der Verbindung conn vor. Der Rückgabewert ist ein Zeiger auf eine Anweisungsstruktur. Die vorbereitete SQL-Anweisung kann anschließend mit Execute ausgeführt werden. Das Datenfeld results der Anweisungsstruktur stat enthält nach dem Aufruf von PrepareStatement Deskriptoren[2] für die Tupel der Ergebnismenge. Wenn die

[2] Deskriptor: beschreibende Datenstruktur

SQL-Anweisung keine Ausgabe liefert, ist results NIL.

- BindParameters (stat, types, numParams) bindet numParams Parameter mit in types beschriebenen Eigenschaften an die Anweisung stat. Parameterplatzerhalter innerhalb von SQL-Anweisungen werden durch ein Fragezeichen dargestellt (z.B. „INSERT INTO TABLE cities (name, population) VALUES (?, ?)“). Die einzelnen Felder von types legen die Eigenschaften der Parameter fest. Die folgende Tabelle soll diese Felder erklären. In den meisten Fällen werden type und mapType gleich sein, außer wenn man automatische Datenkonversion veranlassen möchte oder die Datenbank den Oberondatentyp nicht kennt. Nach dem Aufruf von BindParameters enthält das Datenfeld params der Anweisungsstruktur numParams Felder der entsprechenden Typen, die man über ihren Namen wieder ansprechen kann (FindField(s.params, name)).

Feld von ParamDesc	Beschreibung
type	Oberon-Typ des Parameters (sqlType, prec, scale)
mapType	abgebildeter Datentyp (sqlType, prec, scale)
inOut	Richtung der Datenkommunikation
name	Name des Parameters (für späteren Zugriff)

Tabelle 2.3: Felder von ParamDesc

- Execute(stat) führt die vorbereitete Anweisung stat aus.
- RowCount(stat, rows) liefert in rows die Anzahl der von der SQL-Anweisung (muß entweder eine UPDATE- oder eine DELETE-Anweisung sein) betroffenen Zeilen. Wenn man die Anzahl der von einer SELECT-Anweisung gelieferten Zeilen wissen möchte, muß man eine Anweisung der Art „SELECT COUNT(*) ...“ ausführen (was allerdings ineffizient ist und lange dauern kann, da dazu die gesamte Ergebnismenge der Anweisung berechnet wird und erst anschließend die Anzahl der Zeilen in der Ergebnismenge berechnet und zurückgegeben wird) oder beim Auslesen der Daten einen Zähler mitführen.
- Fetch(stat) überträgt die erste oder nächste Ergebniszeile der bereits ausgeführten SQL-Anweisung stat nach results. Innerhalb der Ergebniszeile results kann man mit den Zugriffsprozeduren FirstField, NextField, PrevField und FindField auf die einzelnen Felder zugreifen.
- ErrS(str, stat) ruft ShowError mit entsprechenden Parameterwerten auf.
- GetStatementOption (stat, option, info) liefert den aktuellen Wert der die SQL-Anweisung stat betreffenden Option option in info. Gültige Werte für option sind im Anhang C.1 zu finden. Abhängig von der Art der Option muß man ein StringField (wenn die Information als Zeichenkette zurückgeliefert wird) oder

ein IntField (wenn die Information als Ganzzahl oder Bitmaske zurückgeliefert wird) zum Auslesen der Werte übergeben.

- stat := Tables(conn) bereitet eine Anweisung vor, über die man alle über die Verbindung conn verfügbaren Tabellen abfragen kann. Die Anweisung stat muß noch (wie eine gewöhnliche Anweisung) ausgeführt werden.

- stat := NewStatement(conn) liefert einen Zeiger auf eine initialisierte Anweisungsstruktur zurück.

- GenerateResultRow(stat) erzeugt die Deskriptoren für die Zeilen der Ergebnismenge. Die SQL-Anweisung stat muß bereits vorbereitet sein.

Die Abbildung 2.3 zeigt das Objektmodell von Anweisungen und Verbindungen. Man sieht die mit einer SQL-Anweisung assoziierte Verbindung, die Parameterzeile und die Ergebniszeile, die beide beliebig viele Felder zur Parameterkommunikation enthalten können.

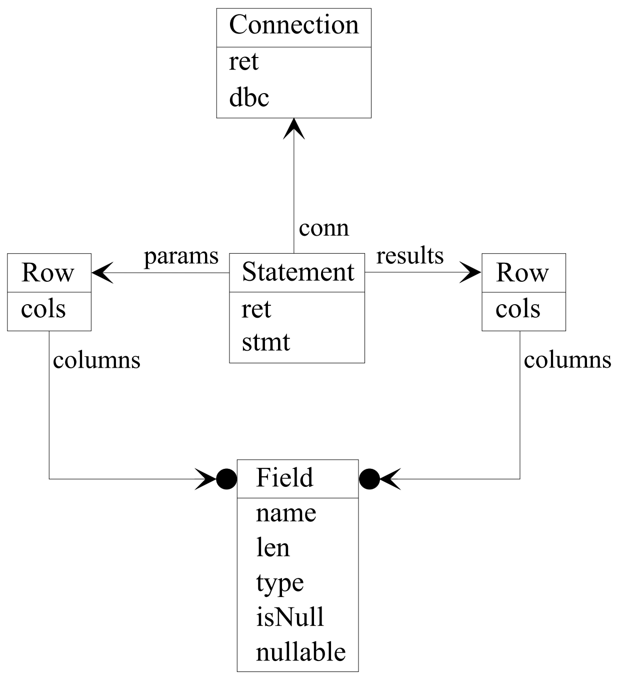

Abbildung 2.3: Objektmodell Statement

2.4.3 Datentypen

Es gibt einige vordefinierte ODBC-Datentypen für Zeichenketten, Zahlen, Datum und Uhrzeit. Datenbanken implementieren meistens nur eine Untermenge der ODBC-

Datentypen. Die von der Datenbank angebotenen Datentypen nennen wir Datenbankdatentypen. Auch die Programmiersprache Oberon unterstützt nicht alle ODBC-Datentypen, sodaß für manche ODBC-Datentypen (wie SQLDate und SQLTime) benutzerdefinierten Datentypen definiert werden müssen.

Datentypen für Zeichenketten

- SQLChar: Zeichenkette fixer Länge (1 bis 254 Zeichen, mit Leerzeichen aufgefüllt), in Oberon ein ARRAY OF CHAR
- SQLVarChar: Zeichenkette variabler Länge mit vorgegebener maximaler Länge (1 bis 254 Zeichen, mit Nullzeichen abgeschlossen), in Oberon ein ARRAY OF CHAR
- SQLLVarChar: lange Zeichenkette variabler Länge mit datenquellenabhängiger maximaler Länge (z.B. 65500 oder 2^{30}), in Oberon eine Datei, die Zeichen enthält
- SQLBin: binäre Daten fixer Länge (1 bis 255 Bytes), in Oberon ein ARRAY OF CHAR (da CHAR vorzeichenlos ist, oder SYSTEM.BYTE)
- SQLVarBin: binäre Daten variabler Länge mit vorgegebener maximaler Länge (1 bis 255 Bytes), in Oberon ein ARRAY OF CHAR
- SQLLVarBin: große binäre Daten variabler Länge mit datenquellenabhängiger maximaler Länge (z.B. 65500 oder 2^{30}), in Oberon eine Datei, die Zeichen enthält

Datentypen für Zahlen

- SQLBit: einzelne binäre Ziffer, in Oberon BOOLEAN
- SQLTinyInt: ganzzahliger Wert im Bereich von -128 bis 127 (mit Vorzeichen) oder 0 bis 255 (ohne Vorzeichen), in Oberon SHORTINT oder INTEGER, wenn ohne Vorzeichen
- SQLSmallInt: ganzzahliger Wert im Bereich von -32768 bis 32767 (mit Vorzeichen) oder 0 bis 65535 (ohne Vorzeichen), in Oberon INTEGER oder LONGINT, wenn ohne Vorzeichen
- SQLInteger: ganzzahliger Wert im Bereich von -2^{31} bis $2^{31}-1$ (mit Vorzeichen) oder 0 bis $2^{32}-1$ (ohne Vorzeichen), in Oberon LONGINT
- SQLBigInt: ganzzahliger Wert im Bereich von -2^{63} bis $2^{63}-1$ (mit Vorzeichen) oder 0 bis $2^{64}-1$ (ohne Vorzeichen), in Oberon nur als Zeichenkette möglich

- SQLDecimal: vorzeichenbehafteter, diskreter numerischer Wert mit prec signifikanten Dezimalstellen ($1 \leq$ prec ≤ 15), davon scale hinter dem Komma ($1 \leq$ scale $\leq$ prec), in Oberon nur als Zeichenkette möglich
- SQLNumeric: wie SQLDecimal
- SQLReal: vorzeichenbehaftete Gleitkommazahl mit einer Mantissenlänge von 7 Dezimalziffern im Bereich von ungefähr $\pm 10^{-38}$ bis $\pm 10^{38}$, in Oberon REAL
- SQLFloat: vorzeichenbehaftete Gleitkommazahl mit einer Mantissenlänge von 15 Dezimalziffern im Bereich von ungefähr $\pm 10^{-308}$ bis $\pm 10^{308}$, in Oberon LONGREAL
- SQLDouble: je nach benötigter Mantissenlänge wie SQLReal oder SQLFloat, in Oberon LONGREAL

Datentypen für Datum und Uhrzeit

- SQLDate: Datum; in Oberon ein Record mit den Feldern year, month und day
- SQLTime: Uhrzeit; in Oberon ein Record mit den Felder hour, minute und second
- SQLTimeStamp: Datum und Uhrzeit; in Oberon ein Record mit den Felder year, month, day, hour, minute, second und fraction

Für die Kommunikation der Daten zwischen der Datenbank und dem Programm wurden verschiedene Klassen definiert. FieldDesc faßt die Eigenschaften zusammen, die jedes Feld hat.

Die spezielleren Felder zur Darstellung von Zeichenketten, Zahlen, Datum und Uhrzeit sind Typerweiterungen von FieldDesc (siehe Abbildung 2.4).

Die einzelnen Felder einer Ergebniszeile und die Parameter einer SQL-Anweisung werden in Zeilen (Row) zusammengefaßt. Eine Row hat als sichtbare Komponente nur cols, das die Anzahl der Felder in der Zeile angibt.

Über die Zugriffsprozeduren FirstField(r, f), NextField(r, f), PrevField(r, f) und FindField(r, name, f) kann man auf die in der Zeile enthaltenen Felder zugreifen. Der Ausgangsparameter f wird auf das erste, nächste, vorige oder zum Namen name passende Feld gesetzt. Mit Typtests und Typzusicherungen kann man auf die Komponenten der erweiterten Felder zugreifen.

2.4.4 Umsetzung von Oberon- auf Datenbankdatentypen

Innerhalb der SQL-Anweisungen müssen die Datenbankdatentypen angegeben werden (z.B. bei einer CREATE-TABLE-Anweisung). Die Abbildung von ODBC-Datentypen auf Datenbankdatentypen erfolgt aber nicht automatisch durch den ODBC-Treibermanager, sondern mit der Funktion Map(conn, type, mapType). Wenn die

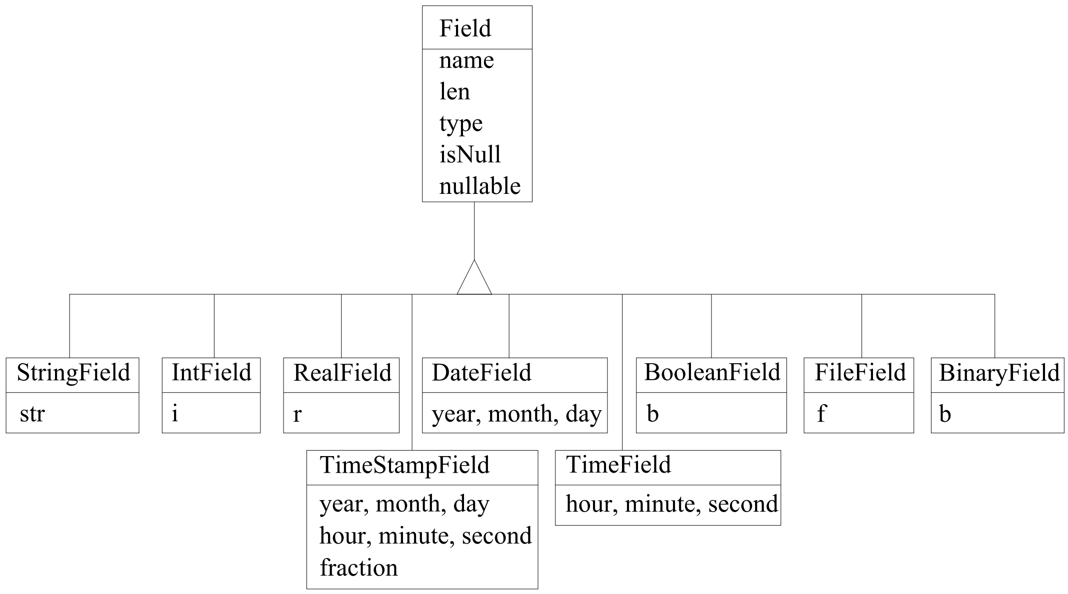

Abbildung 2.4: FieldDesc mit Typerweiterungen

Umsetzung mit Map gelungen ist, erhält man mit GetTypeName den Namen und die Parameter des Typs als Zeichenkette. Es gibt je nach Datenbank nicht nur Unterschiede in der Menge der Datenbankdatentypen, sondern auch Unterschiede in den Parametern bei der Datendefinition. Der Datenbankdatentyp für eine Zeichenkette ist unter Oracle z.B. CHAR(32), unter MS Access aber bloß TEXT ohne weitere Parameter.

Auch bei Abfragen von Daten aus der Datenbank muß man wieder unterschiedliche Datentypen beachten. Die Datenfelder werden automatisch anhand der Datenbankdatentypen angelegt und können sich in ihren Typen von den beim Einfügen von Daten verwendeten Feldern unterscheiden (siehe Abbildung 2.5 und Abbildung 2.6). Wenn man z.B. einen ganzzahligen Wert (LONGINT) in eine Oracle Datenbank einfügt, muß man als Datenbankdatentyp NUMBER verwenden. LONGINT wird dabei auf den ODBC-Datentyp SQLInteger abgebildet, der wiederum auf den Datenbankdatentyp NUMBER. Beim Auslesen der Daten wird der Datenbankdatentyp NUMBER allerdings auf den ODBC-Datentyp SQLChar beziehungsweise ARRAY OF CHAR abgebildet, weil es im allgemeinen nicht möglich ist, den Datenbankdatentyp NUMBER (Fixpunktzahl) mit den normalen numerischen Datentypen einer Programmiersprache darzustellen.

Die Umsetzung von Oberon-Datentypen auf Datenbankdatentypen (und vice versa) ist also im allgemeinen nicht ohne Informationsverlust möglich (siehe Abbildung 2.6). Die Typen der Felder zur Datenkommunikation können sich also beim Einfügen von Daten und beim Auslesen unterscheiden (vgl. LongIntValueOf in AppendToBench1 im Beispielprogramm ODBCBench im Anhang B.2).

Erweiterung	für folgende SQL-Datentypen
StringField	SQLChar, SQLVarChar, SQLNumeric, SQLDecimal, SQLBigInt
IntField	SQLTinyInt, SQLSmallInt, SQLInteger
RealField	SQLFloat, SQLReal, SQLDouble
BooleanField	SQLBit
DateField	SQLDate
TimeField	SQLTime
TimeStampField	SQLTimeStamp
FileField	SQLLVarChar, SQLLVarBin
BinaryField	SQLBin, SQLVarBin

Tabelle 2.4: Fields und entsprechende SQL-Datentypen

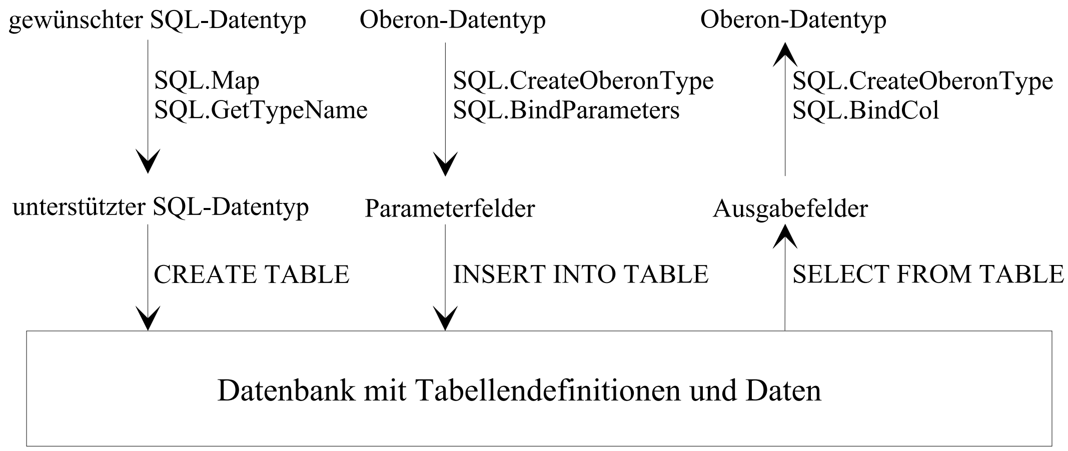

Abbildung 2.5: Umsetzung von Oberon- auf Datenbankdatentypen

2.4.5 Dateibasierte Datentypen

Die ODBC-Datentypen SQLLVarChar und SQLLVarBin sind Datentypen, für die man bei der Datenkommunikation nicht einfach genügend große Puffer anlegen kann. Wie schon oben erwähnt, können solche Datenfelder nahezu uneingeschränkt groß werden. Verwendet werden diese Datentypen z.B. für BLOBs (binary large objects) wie Bilder oder OLE-Objekte innerhalb von Tabellen. Die Werte solcher Datenfelder werden stückweise übertragen (in Blöcken von je einem Kilobyte). Als Puffer für solche Datenfelder werden temporäre Dateien verwendet, die beliebig groß werden können. Beim Einfügen von dateibasierten Datentypen in eine Datenbank muß man vor der Ausführung die zu übertragenden Daten in die temporäre Datei (Komponente f des FileFields) schreiben, beim Abfragen werden die Daten in Fetch in die temporäre Datei geschrieben.

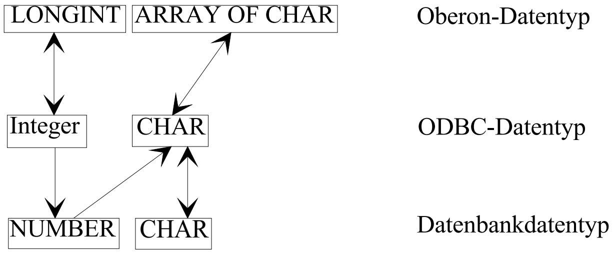

Abbildung 2.6: Informationsverlust bei Umsetzung von Datentypen

2.4.6 Beispiele

„Static SQL"-Anweisungen

Beliebige SQL-Anweisungen, die keine Parameter-Platzhalter enthalten (auch statisches SQL genannt), kann man einfach vorbereiten und ausführen. Beispiele dafür sind das Anlegen einer neuen Tabelle, das Einfügen von Datensätzen (wenn man die gesamte SQL-Anweisung inklusive den Werten der einzelnen Attribute als Zeichenkette vorliegen hat) und das Löschen einer Tabelle.

```
stat := SQL.PrepareStatement(conn, "CREATE TABLE Addresses (firstName TEXT(32),
   lastName TEXT(32), street TEXT(32), town TEXT(32), zipCode LONG)");
SQL.Execute(stat);
stat := SQL.PrepareStatement(conn, "INSERT INTO Addresses VALUES
   ('Christoph', 'Steindl', 'Freistädterstraße 315', 'Linz', 4040)");
SQL.Execute(stat);
stat := SQL.PrepareStatement(conn, "DROP TABLE Addresses");
SQL.Execute(stat);
```

SQL-Anweisungen mit Parameter-Platzhaltern

Für Parameter von SQL-Anweisungen kann an Platzhalter angeben, die bei der Ausführung durch Variablenwerte ersetzt werden. Solche Platzhalter werden durch „?" ausgedrückt, z.B.:

```
stat := SQL.PrepareStatement(conn, "INSERT INTO Friends VALUES (?, ?, ?, ?, ?)");
```

vor der Ausführung muß der Programmierer an jeden Platzhalter eine Variable binden. Anschließend kann er den Variablen Werte zuweisen und die Anweisung mittels SQL.Execute ausführen.

Variablen können mittels SQL.BindParameters an Platzhalter gebunden werden. Dabei muß für jeden Platzhalter ein Record vom Typ SQL.ParamDesc aufgebaut werden, das den Typ des Parameters beschreibt, z.B.:

```
VAR types: ARRAY 5 OF SQL.ParamDesc; (* description of placeholders *)
...
```

```
(* assume that parameter 0 is a string *)
types[0].type := stringType; types[0].mapType := stringMapType;
types[0].inOut := SQL.InParam; types[0].name := "firstName";
... do the same for the other parameters
SQL.BindParameters(stat, types, 5);
```

stringType und stringMapType müssen an geeigneter Stelle (z.B. nach dem Aufbau der Verbindung zur Datenbank) initialisiert werden. Wenn der Datentyp stringType umgesetzt werden konnte, liefert SQL.Map TRUE, und stringMapType ist der umgesetzte Datentyp, z.B.:

```
stringType.sqlType := SQL.OberonChar; stringType.prec := 32; stringType.scale := 0;
IF ~SQL.Map(conn, stringType, stringMapType) THEN HALT(69) END ;
```

Nun können den Parametern Werte zugewiesen werden. SQL.BindParameters legt für jeden Parameter ein Feld vom Typ SQL.Field an. Die Parameterwerte müssen in diese Felder kopiert werden, z.B.:

```
VAR field: SQL.Field; firstName: ARRAY 32 OF CHAR;
...
SQL.FindField(stat.params, "firstName", field); (* get field of parameter 0 (firstName) *)
COPY(firstName, field(SQL.StringField).str);
```

Wenn alle Felder mit Werten belegt sind, kann die Anweisung schließlich ausgeführt werden:

```
SQL.Execute(stat)
```

Folgendes Beispiel zeigt diese Aktionen nochmals in ihrem Zusammenhang:

```
(* global variables *)
VAR
  stringType, stringMapType, longintType, longintMapType: SQL.Type;
  ...

  (* after connecting to the database *)
  stringType.sqlType := SQL.OberonChar; stringType.prec := 32; stringType.scale := 0;
  IF ~SQL.Map(conn, stringType, stringMapType) THEN HALT(69) END ;

  longintType.sqlType := SQL.OberonLongInt; longintType.prec := 0; longintType.scale := 0;
  IF ~SQL.Map(conn, longintType, longintMapType) THEN HALT(69) END ;
  ...

PROCEDURE Insert*;
  VAR firstName, lastName, street, town: ARRAY 32 OF CHAR; zipCode: LONGINT;
```

```
    stat: SQL.Statement; field: SQL.Field; types: ARRAY 5 OF SQL.ParamDesc;
BEGIN
  In.Open; In.String(firstName); In.String(lastName); In.String(street);
  In.String(town); In.LongInt(zipCode);
  stat := SQL.PrepareStatement(conn, "INSERT INTO Friends VALUES (?, ?, ?, ?, ?)");
  types[0].type := stringType; types[0].mapType := stringMapType;
  types[0].inOut := SQL.InParam;
  types[1] := types[0]; types[2] := types[0]; types[3] := types[0];
  types[0].name := "firstName"; types[1].name := "lastName";
  types[2].name := "street"; types[3].name := "town";
  types[4].type := longintType; types[4].mapType := longintMapType;
  types[4].inOut := SQL.InParam;
  types[4].name := "zipCode";
  SQL.BindParameters(stat, types, 5);
  SQL.FindField(stat.params, "firstName", field);
  COPY(firstName, field(SQL.StringField).str);
  SQL.FindField(stat.params, "lastName", field);
  COPY(lastName, field(SQL.StringField).str);
  SQL.FindField(stat.params, "street", field);
  COPY(street, field(SQL.StringField).str);
  SQL.FindField(stat.params, "town", field);
  COPY(town, field(SQL.StringField).str);
  SQL.FindField(stat.params, "zipCode", field);
  field(SQL.IntField).i := zipCode;
  SQL.Execute(stat)
END Insert;
```

Abfragen

Bei Abfragen von Daten aus Tabellen muß für jede Spalte in der Ergebnismenge ein Feld für die Datenkommunikation angelegt werden. Das passiert automatisch beim Vorbereiten der SQL-Anweisung, stat.results enthält dann einen Zeiger auf die Ergebniszeile mit den entsprechenden Feldern oder NIL, wenn keine Ergebnisse zu erwarten sind. Über diese Felder kann man mit den Aufrufen FirstField(stat.results, field und NextField(field) iterieren, bis field den Wert NIL hat. Jedes Feld der Ergebniszeile hat (wie die Parameterfelder) einen Namen. Dieser Name ist der Name der Spalte (wie in der selectList der SELECT-Anweisung oder bei der Tabellendefinition angegeben) und wird im folgenden Beispiel vor der zeilenweisen Ausgabe der eigentlichen Daten als Überschrift für jede Spalte der Ergebniszeile ausgegeben. Die Ergebniszeilen selber werden nacheinander mit Fetch ausgelesen und mit der Hilfsprozedur PutField ausgegeben.

```
PROCEDURE ListAll*;
  VAR stat: SQL.Statement; field: SQL.Field;
BEGIN
```

```
  stat := SQL.PrepareStatement(conn, "SELECT * FROM Friends");
  SQL.Execute(stat);
  IF (stat.ret = SQL.Success) & (stat.results # NIL) THEN
    Out.Ln; SQL.FirstField(stat.results, field);
    WHILE field # NIL DO
      Out.String(field.name); Out.Char(" ");
      SQL.NextField(field)
    END ;
    REPEAT
      SQL.Fetch(stat);
      IF stat.ret = SQL.Success THEN
        Out.Ln;
        SQL.FirstField(stat.results, field);
        WHILE field # NIL DO
          PutField(field); Out.Char(" ");
          SQL.NextField(field)
        END
      END
    UNTIL stat.ret # SQL.Success
  END
END ListAll;
```

2.5 SQLExt

Das Modul SQLExt bietet Erweiterungen zum Modul SQL an, die den ODBC Konformitätsebenen 1 und 2 entsprechen. Das vollständige Definitionsmodul befindet sich im Anhang A.4.

Die Funktionen Columns, ColumnPrivileges, ForeignKeys, PrimaryKeys, TablePrivileges, Procedures, ProcedureColumns, SpecialColumns und Statistics liefern als Rückgabewert eine vorbereitete Anweisungsstruktur. Man kann diese Anweisungen anschließend ausführen und die Ergebnisse wie bei normalen SQL-Anweisungen bearbeiten (auf die einzelnen Felder der Ergebniszeilen zugreifen etc.). Die Namen der Funktionen deuten schon darauf hin, was die Ergebnismengen enthalten. Genaue Beschreibungen findet man in [ODBC94], Kurzbeschreibungen findet man im Anhang A.4 und erhält man auch durch die Ausführung des Kommandos Def.Show SQLExt.Mod.

An die Zeichenketten-Parameter kann man, wie erwartet, Zeichenketten übergeben, aber zum Beispiel auch den im Modul SQL definierten NullString (SQL.nullString) oder, wenn der formale Parametername auf „Pat" (für engl. pattern = Suchmuster) endet, eine Zeichenkette mit den speziell behandelten Zeichen „%" (steht für beliebig viele beliebige Zeichen, entspricht dem Stern unter MS-DOS) und „_" (steht für ein beliebiges Zeichen, entspricht dem Fragezeichen unter MS-DOS). Die spezielle Behandlung des NullStrings ist notwendig, weil die Datenbanktreiber unterschiedlich reagieren, wenn man einen Zeiger auf eine leere Zeichenkette oder einen NIL-Zeiger als Aktualparameter übergibt. In der Regel muß man SQL.nullString verwenden, wenn man zum Beispiel die Namen der interessierenden Tabellen bei einem Aufruf von Tables nicht kennt.

Manche Datenbanktreiber unterstützen positioniertes Ändern und Löschen von Datensätzen. Dazu führt man zuerst eine Abfrage der Art "SELECT * FROM Adresses FOR UPDATE" aus, geht dann zum Datensatz, den man ändern oder löschen will (durch wiederholte Aufrufe von Fetch), und führt entweder die Anweisung „UPDATE Addresses SET zipCode = ? WHERE CURRENT OF cursor“ oder „DELETE FROM Addresses WHERE CURRENT OF cursor“ aus. Da unter ODBC die Cursor automatisch angelegt werden und man deren Namen nicht kennt, man sie aber in der Zeichenkette der SQL-Anweisung angeben muß, gibt es die Prozeduren GetCursorName und SetCursorName, mit denen man den Namen des Cursors einer SQL-Anweisung auslesen und ändern kann. cursor wäre in den oben angegebenen SQL-Anweisungen der für die SELECT-Anweisung von GetCursorName gelieferte Name.

Die Prozedur AppendType(conn, str, type) kann man verwenden, um die Zeichenkette für den Oberon-Datentyp type an die Zeichenkette str anzuhängen. Der Oberon-Datentyp type wird dazu abgebildet auf den entsprechenden Datenbankdatentyp, dessen Name (einschließlich der notwendigen Parameter) an die Zeichenkette angehängt werden.

Das folgende Programstück erstellt die Zeichenkette einer CREATE-TABLE-Anweisung.

```
  VAR int, str: SQL.Type; s: ARRAY 256 OF CHAR;
BEGIN
  int.sqlType := OberonLongInt; int.prec := 0; int.scale := 0;
  str.sqlType := OberonChar; str.prec := 32; str.scale := 0;
  s := "CREATE TABLE Test (i ";
  SQLExt.AppendType(conn, s, int);
  Strings.Append(", name ", s);
  SQLExt.AppendType(conn, s, str);
  Strings.Append(")", s)
  ...
END
```

Die restlichen Prozeduren wurden der Vollständigkeit halber aufgenommen und werden hier nicht weiter beschrieben.

2.6 DBBrowser

Der DBBrowser ist ein Anwendungsprogramm, das für den Datenbankadministrator gedacht ist und die Administration von Datenbanken unterstützt.

Nachdem man eine Verbindung zu einer Datenbank aufgebaut hat, kann man mit ihm beliebige SQL-Anweisungen ausführen. Weiters kann man mit dem DBBrowser alle Prozeduren der Module SQL und SQLExt für die aktuelle Verbindung aufrufen. Die Ergebnisse werden tabellenartig angezeigt, sind aber nicht editierbar.

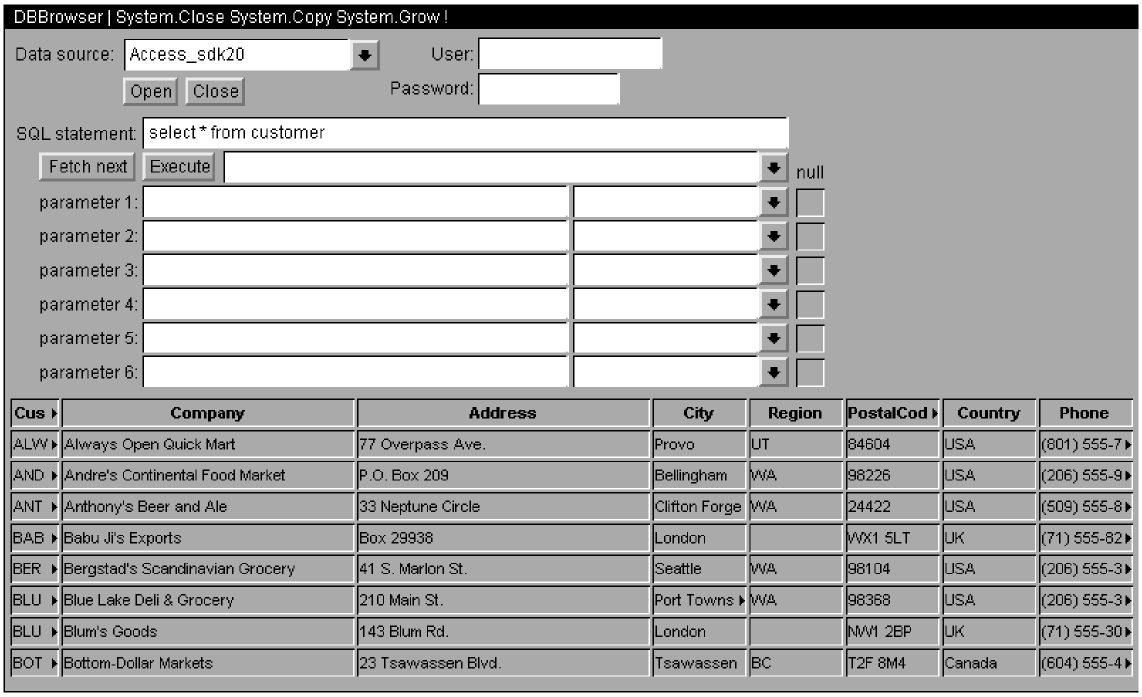

Abbildung 2.7: DBBrowser-Dialog

Einige Beispiele sollen das Arbeiten mit dem DBBrowser demonstrieren:

Aufbauen einer Verbindung

Zum Aufbauen der Verbindung wählt man aus der Liste „Data source“ den gewünschten ODBC-Datenbanktreiber aus, trägt in die Textfelder „User“ und „Password“ die Benutzerkennung und das Paßwort ein und klickt auf den Open-Knopf. Wenn beim Aufbauen der Verbindung ein Fehler aufgetreten ist, wird eine Fehlermeldung im System.Log ausgegeben. Benötigt der Datenbanktreiber weitere Informationen zum Verbindungsaufbau (oder hat man die Felder „User“ und „Password“ nicht ausgefüllt), erfragt der Datenbanktreiber diese mit eigenen Dialogen.

Schließen einer Verbindung

Nach Abschluß seiner Arbeit kann man die Verbindung durch einen Klick auf den Close-Knopf wieder schließen. Macht man das nicht, wird die Verbindung, nachdem man seine Arbeiten mit dem DBBrowser beendet hat, automatisch geschlossen.

Ausführen einer SQL-Anweisung

Zum Ausführen von SQL-Anweisungen schreibt man die SQL-Anweisung (z.B. „SELECT * FROM Addresses") in das Textfeld „SQL statement" und klickt auf den Execute-Knopf. Wenn die Anweisung Parameter-Platzhalter hat, muß man für jeden Parameter den Wert in das Textfeld „parameter i" eintragen und den zu verwendenden Datenbankdatentyp aus der Liste daneben auswählen. Man kann den Wert des Parameters als Nullwert spezifizieren, indem man die Auswahlbox „null" anwählt. Die Ergebnisse der Abfrage werden tabellenartig ausgegeben (wie in der Abbildung 2.7 unter den Parameter-Textfeldern sichtbar). Man kann in dieser Tabelle nicht direkt editieren. Ein Ergebnisdatensatz entspricht einer Zeile. Die einzelnen Zellen werden solange verkleinert, bis alle in einer Zeile Platz haben. Wenn der Inhalt einer Zelle nicht vollständig sichtbar ist, kann man nach links und rechts scrollen, indem man auf die kleinen schwarzen Dreiecke am linken bzw. rechten Rand der Felder klickt. Vertikales Scrollen erreicht man, indem man die nächsten Datensätze von der Datenbank anfordert (Fetch-next-Knopf).

Ausführen der in SQL und SQLExt definierten Prozeduren

Man kann alternativ zu SQL-Anweisungen auch Kommandos aus der Liste neben dem Execute-Knopf auswählen. Diese entsprechen den Prozeduren aus den Modulen SQL und SQLExt und haben jeweils die dort angeführten Parameter. Für diese Parameter muß man wiederum vor der Ausführung des Kommandos Werte in den Textfeldern „parameter i" eintragen. Es wird immer zuerst versucht, eine SQL-Anweisung auszuführen (wenn eine angegeben ist), weshalb es notwendig sein kann, den Text im Textfeld „SQL statement" zu löschen.

Ausgabe aller Datenbankdatentypen

Um alle von der aktuellen Verbindung unterstützten Datenbankdatentypen auszugeben, wählt man aus der Liste neben dem Execute-Knopf „DataTypes sqlType" aus und schreibt in das Textfeld „parameter 1:" die Nummer des ODBC-Datentyps (1 für SQLChar oder OberonChar, 5 für OberonInteger etc., 0 für die Ausgabe aller Datenbankdatentypen). Ausgegeben werden dann alle Datenbankdatentypen, die den gewünschten ODBC-Datentyp implementieren. Anschließend klickt man auf den Execute-Knopf, weitere Daten kann man durch Klicken auf den Fetch-next-Knopf anzeigen (wenn vorhanden).

Ausgabe aller Tabellen

Um alle über die aktuelle Verbindung zugänglichen Tabellen auszugeben, wählt man aus der Liste neben dem Execute-Knopf „Tables tableQual tableOwnerPat tablePat tableTypes" aus, trägt in die 4 Parameter-Textfelder die gewünschten Werte ein und klickt auf den Execute-Knopf. Wenn man nicht weiß, was man als „table qualifier",

„table owner“, „table name“ oder „table type“ eintragen soll, setzt man die Auswahlfelder „null“ rechts von den Parameter-Textfeldern. Die Ausgabe zeigt dann alle möglichen Tabellen mit den entsprechenden Werten für „table qualifier“, „table owner“, „table name“ und „table type“.

Anzeige von Optionen der Verbindung

Um die aktuellen Werte von Optionen anzuzeigen, wählt man aus der Liste neben dem Execute-Knopf „GetConnectionOption integerOptionNumber“ aus, trägt im ersten Parameter-Textfeld die Nummer der gewünschten Option ein und klickt auf den Execute-Knopf. Die Nummern der Optionen findet man im Anhang C.2. Um z.B. herauszufinden, ob der Datenbanktreiber automatisch nach jeder Ausführung einer SQL-Anweisung ein Commit macht, kann man GetConnectionOption für die Option mit der Nummer 102 (SQL_AUTOCOMMIT) aufrufen. Die Ausgabe ist dann einer der Werte, die im Anhang C.2 nach SQL_AUTOCOMMIT angeführt sind.

Einfügen von Daten in eine Tabelle

Mit einer INSERT-Anweisung kann man einen Datensatz in die Tabelle eintragen. Dazu führt man die INSERT-Anweisung ohne Parameter direkt aus (zum Beispiel „INSERT INTO TestTable VALUES ('Christoph', 100, '1996-01-15 12:15:0.0')“, oder verwendet eine INSERT-Anweisung mit Parametern, um mehrere Datensätze mit unterschiedlichen Werten einzufügen (zum Beispiel „INSERT INTO TestTable VALUES (?, ?, ?)“). Dazu schreibt man in den Parameter-Textfeldern die Werte und wählt aus den Auswahllisten daneben den entsprechenden Datenbankdatentypen aus (CHAR für OberonChar, INTEGER für OberonLongInt und DATETIME für OberonDateTime).

2.7 DataEditor

Der DataEditor ist ein Anwendungsprogramm, das für den Endbenutzer gedacht ist, der damit seine Tabellen datensatzweise bearbeitet. Man kann neue Datensätze einfügen, bestehende ändern oder löschen und nach Datensätzen suchen.
Mit den Knöpfen „First“ und „Next“ kann man durch die Datensätze blättern, „New“ erstellt einen neuen, leeren Datensatz und „Update“ ändert die Daten des aktuellen Datensatzes auf die angezeigten Werte. Wenn man zuvor einen neuen Datensatz erzeugt hat, wird dieser durch ein „Update“ in die Tabelle eingefügt beziehungsweise durch ein „Delete“ wieder verworfen.

Mit „Search“ kann man nach Datensätzen suchen. In der Regel erzeugt man sich dazu zuerst einen leeren Datensatz (oder löscht aus einem dargestellten Datensatz alle nicht interessierenden Felder) und schreibt dann in die Felder, die als Suchkriterien dienen sollen, die gewünschten Werte. Alle Felder, die nicht leer sind, werden

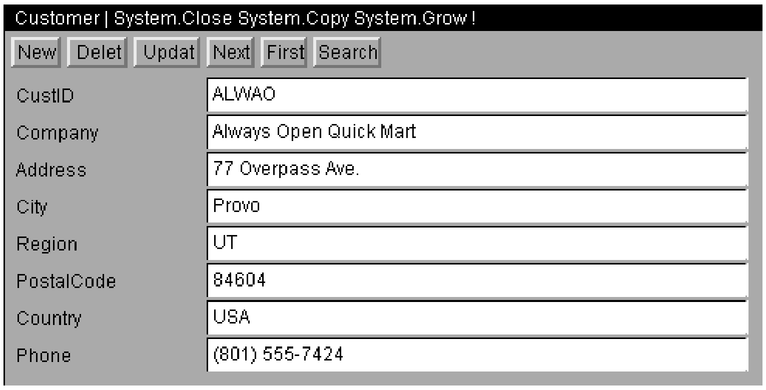

Abbildung 2.8: DataEditor-Dialog

zu einem gemeinsamen Suchkriterium UND-verknüpft. Man kann am Beginn des Feldes die Vergleichsoperatoren `<`, `<=`, `>`, `>=`, `#` und LIKE angeben. Zeichenketten können, wenn mit LIKE gesucht wird, die speziell behandelten Zeichen „%“ und „_“ (siehe Seite 52) enthalten. Mit einem Klick auf den „Search“-Knopf sucht man nach allen Datensätzen, die den Suchkriterien entsprechen. Man kann anschließend durch diese Datensätze blättern, kommt aber mit „First“ wieder zurück zum ersten aller Datensätze.

Kapitel 3

Stücklistenverwaltung

In diesem Kapitel wird gezeigt, wie man mit relationaler Datenmodellierung eine Stücklistenverwaltung implementieren kann.

3.1 Relationale Datenmodellierung

3.1.1 Teil und Teilestrukturen

Ein Teil kann aus mehreren anderen Teilen bestehen (ist ein Oberteil) beziehungsweise kann ein Teil in mehrere andere Teile mit einer bestimmten Menge eingehen (ist ein Unterteil). Bei einem Teil kann es sich um einen Eigenteil, einen Fremdteil oder um einen verkaufsfähigen Teil (Artikel) handeln. Es gibt sowohl einige Eigenschaften, die alle Teile aufweisen (Teilenummer, Bezeichnung, Art, Lagerstand, Mengeneinheit), als auch für jeden Teil spezielle Eigenschaften (zum Beispiel für Verkaufsteile Verkaufspreis, für Fremdteile Bestellpolitik und für Eigenteile Losgröße). Nur der Eigenteil muß in Form von Fertigungsaufträgen produziert werden.

3.1.2 Betriebsmittel, technische Verfahren und Arbeitspläne

Ein Eigenteil wird nach bestimmten technischen Verfahren mit den benötigten Komponenten auf bestimmten Betriebsmitteln gefertigt. Diese Daten werden im Arbeitsplan festgehalten. Das technische Verfahren beschreibt die grundsätzlich einzusetzenden Fertigungsverfahren wie Bohren, Drehen und Fräsen. Das technische Verfahren ist durch eine Verfahrensnummer gekennzeichnet und durch eine Reihe von technischen Attributen beschrieben.

Um eine Komponente herzustellen, wird von der Arbeitsvorbereitung eine bestimmte Reihenfolge auszuführender technischer Verfahren zusammengestellt, die im Arbeitsplan festgehalten werden. Ein Arbeitsplan besteht aus einigen Kopfdaten wie Name des Erstellers, Erstellungsdatum usw. und der Beschreibung der einzelnen Arbeitsgänge. Ein Arbeitsgang erfordert die Angabe des technischen Verfahrens, der benötigten Teile mit deren Menge, der verwendeten Betriebsmittel und der Rüstund Arbeitszeit. Ein technisches Verfahren kann in mehreren Arbeitsgängen angewendet werden.

Es ist möglich, einen Arbeitsgang auf mehreren Betriebsmitteln und auf einem Betriebsmittel verschiedene Arbeitsgänge zu fertigen („Arbeitsgangzuordnung“), wobei hier besonders die Rüst- und Arbeitszeiten interessante Attribute darstellen.

Komponenten, die zur Herstellung eines Teils erforderlich sind, können in unterschiedlichen Arbeitsgängen (verschiedene Arbeitspläne) benötigt werden und umgekehrt in einem Arbeitsgang mehrere Komponenten („Arbeitsgangkomponentenzuordnung“).

Für einen Eigenteil können verschiedene Arbeitspläne („Arbeitsplanzuordnung“) bestehehen, zum Beispiel einer, bei dem ein hochtechnisiertes Fertigungsverfahren angewendet wird, und ein anderer für weniger technisierte Fertigungsanlagen mit geringeren Rüstkosten. Arbeitspläne für den gleichen Teil können sich auch nach verschiedenen Betriebsstandorten unterscheiden. Andererseits können fertigungstechnisch ähnliche Teile innerhalb der Stücklistenstruktur nach dem gleichen Arbeitsplan gefertigt werden (zum Beispiel, wenn sich Teile nur durch den Einsatz unwesentlich verschiedener Materialien, wie verschiedene Farben, unterscheiden).

3.1.3 Entity-Relationship Modell

In der Abbildung 3.1 ist die Modellierung der Stücklistenverwaltung durch Entitäten und Beziehungen dargestellt.

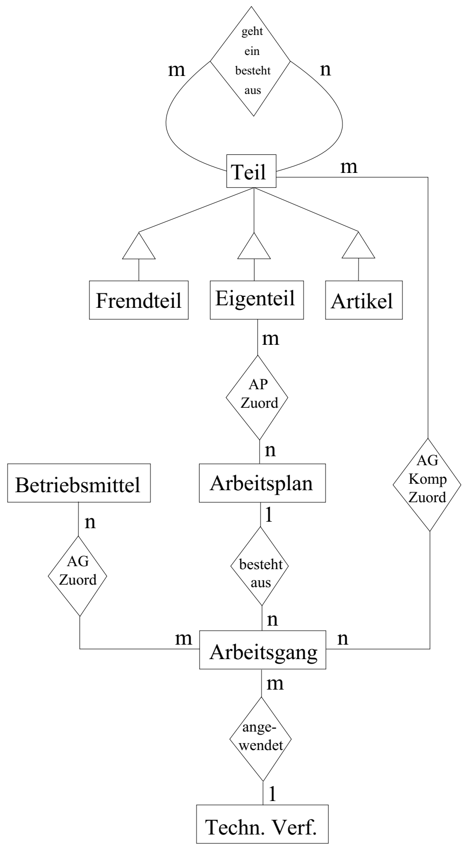

Abbildung 3.1: ER-Diagramm für Stücklistenverwaltung

3.1.4 Relationen in 3. Normalform

Bei der relationalen Modellierung versucht man, die Relationen zur Vermeidung von Redundanz in Normalformen zu bringen (siehe Kapitel 1.2.1).

Die einzelnen Relationen der Stücklistenverwaltung haben schlußendlich folgende Attribute:
Teile: Nummer, Bezeichnung, Art, Lagerbestand, Einheit, ...
Struktur: Oberteil-Nummer, Unterteil-Nummer, Menge
Verkaufsteile: Nummer, Verkaufspreis, ...
Fremdteile: Nummer, Bestellpolitik, ...
Eigenteile: Nummer, Losgröße, ...
Arbeitsplan: APNummer, Ersteller, Prüfer, Erstellungsdatum, ...
Arbeitsplanzuordnung: TeileNummer, APNummer, ...
Technisches Verfahren: VNummer, Bezeichnung, ...
Arbeitsgang: APNummer, AGNummer, VNummer, ...
Betriebsmittel: BMNummer, Bezeichnung, Wartung, ...
Arbeitsgangzuordnung: APNummer, AGNummer, BMNummer, ...
Arbeitsgangkomponentenzuordnung: APNummer, AGNummer, Oberteil-Nummer, Unterteil-Nummer, ...

Aus diesen Relationen kann man die Informationen zu Mengenübersichts-, Struktur- und Baukastenstücklisten sammeln.

Die Mengenübersichtsstückliste enthält für einen konkreten Teil alle zur Herstellung benötigten Teile.

Mengenübersichtsstückliste		
1 <Mengeneinheit> <Bezeichnung>		
Teilenr.	Bezeichnung	Menge

Abbildung 3.2: Mengenübersichtsstückliste

Die Strukturstückliste enthält zusätzlich die Information der Schachtelungstiefe.

Strukturstückliste			
1 <Mengeneinheit> <Bezeichnung>			
Stufe	Teilenr.	Bezeichnung	Menge

Abbildung 3.3: Strukturstückliste

Die Baukastenstückliste enthält für einen konkreten Teil alle zur Herstellung benötigten Teile, ohne diese aber wieder in deren Unterteile aufzusplitten.

Baukastenstückliste		
1 <Mengeneinheit> <Bezeichnung>		
Teilenr.	Bezeichnung	Menge

Abbildung 3.4: Baukastenstückliste

Kapitel 4

Implementierungsaspekte

4.1 Abbildung von Datentypen

Die Funktion SQL.Map implementiert die Umsetzung von gewünschten auf vom Datentreiber unterstützte Datentypen. Abbildung 4.1 zeigt, wie versucht wird, nicht-unterstützte Datentypen zu emulieren. In der Graphik deutet ein Pfeil an, daß der Datentyp, auf den der Pfeil zeigt, für den jeweils anderen Datentyp verwendet werden kann. Es ist ohne Informationsverlust möglich, Integer anstatt SmallInt, TinyInt oder Bit zu verwenden. Ebenso kann man Double anstatt von Float und Real verwenden. Weiters ist es immer möglich, numerische Werte oder Uhrzeit und Datum in Zeichenketten umzuwandeln. Kurze Zeichenkettendatentypen können wieder durch längere ersetzt werden. Kurze binäre Datentypen können durch längere ersetzt werden, nicht aber durch Zeichenkettendatentypen. Unter gewissen Umständen ist es möglich, den Datentyp Numeric durch SmallInt oder Integer zu ersetzen: wenn Numeric ohne Nachkommastellen definiert wurde und die Anzahl der signifikanten Stellen den Wertebereich von SmallInt oder Integer nicht übersteigt. Bei der Umsetzung wird darauf geachtet, daß es deshalb keine Endlosschleife gibt. Wenn man schlußendlich bei LVarChar oder LVarBin angelangt ist und auch diese Datentypen nicht unterstützt werden, kann der gewünschte Datentyp nicht abgebildet werden und Map liefert FALSE zurück.

Die Prozedur Map versucht nun in einer LOOP-Schleife, einen Datenbankdatentyp zu finden, der dem gewünschten Datentyp entspricht.

Im Pseudocode sieht die Funktion wie folgt aus:

```
PROCEDURE Map* (c: Connection; type: Type; VAR mapType: Type): BOOLEAN;
BEGIN
  mapType.sqlType := 0; mapType.prec := 0; mapType.scale := 0;
  LOOP
    CASE type.sqlType OF
    SQLBit:
      IF c supports SQLBit THEN mapType.sqlType := SQLBit; RETURN TRUE
      ELSE type.sqlType := SQLTinyInt
```

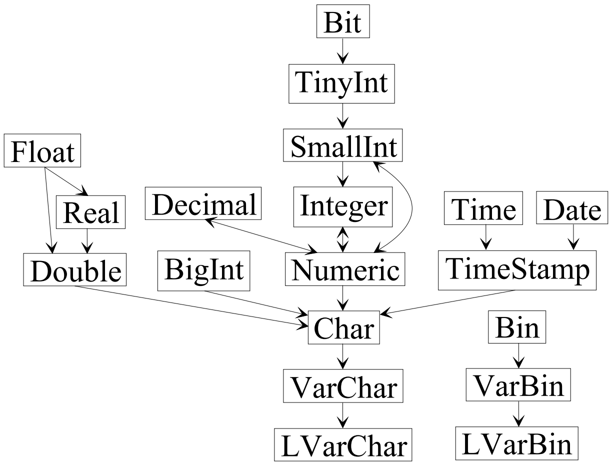

Abbildung 4.1: Abbildung von Datentypen

```
  END
| SQLTinyInt:
  IF c supports SQLTinyInt THEN mapType.sqlType := SQLTinyInt; RETURN TRUE
  ELSE type.sqlType := SQLSmallInt
  END
| SQLBigInt:
  IF c supports SQLBigInt THEN mapType.sqlType := SQLBigInt; RETURN TRUE
  ELSE type.sqlType := SQLChar; prec := 20
  END
| SQLLVarChar:
  IF c supports SQLLVarChar with prec THEN
    mapType.sqlType := SQLLVarChar; mapType.prec := type.prec; RETURN TRUE
  ELSE RETURN FALSE
  END
| SQLVarChar:
  IF c supports SQLVarChar with prec THEN
    mapType.sqlType := SQLVarChar; mapType.prec := type.prec; RETURN TRUE
  ELSE type.sqlType := SQLLVarChar
  END
| SQLChar:
  IF c supports SQLChar with prec THEN
    mapType.sqlType := SQLChar; mapType.prec := type.prec; RETURN TRUE
  ELSE type.sqlType := SQLVarChar
  END
| SQLNumeric:
  IF c supports SQLNumeric with prec and scale THEN
    mapType.sqlType := SQLNumeric; mapType.prec := type.prec;
    mapType.scale := type.scale; RETURN TRUE
```

```
      ELSIF scale = 0 THEN
        IF type.prec < 5 and c supports SQLSmallInt THEN type.sqlType := SQLSmallInt
        ELSIF type.prec < 10 and c supports SQLInteger THEN type.sqlType := SQLInteger
        ELSE type.sqlType := SQLChar; type.prec := type.prec + 2 (* for comma and 0X *)
        END
      ELSE type.sqlType := SQLChar; type.prec := type.prec + 2 (* for comma and 0X *)
      END
    | SQLInteger:
      IF c supports SQLInteger THEN mapType.sqlType := SQLInteger; RETURN TRUE
      ELSE type.sqlType := SQLNumeric; type.prec := 10; type.scale := 0
      END
    ...
    END
  END Map;
```

4.2 Ausführen von Anweisungen

Beim Ausführen von SQL-Anweisungen mit Parametern müssen einige Fälle berücksichtigt werden: für Parameter, die Nullwerte enthalten, muß die Komponente len auf NullData gesetzt werden, für Zeichenkettenparameter auf NTS (null-terminated string), für dateibasierte Parameter schließlich auf ODBC.DataAtExec, da in diesem Fall die Daten blockweise beim Aufruf von Execute übertragen werden. Dieses blockweise Übertragen geschieht nach dem Aufruf von ODBC.SQLExecute für alle dateibasierten Parameterfelder, solange ODBC.SQLParamData NeedData liefert.

```
PROCEDURE Execute* (s: Statement);
  VAR r: Files.Rider; ret: ODBC.Retcode; field: Field; dummy: LONGINT;
BEGIN
  ...
  FirstField(s.params, field);
  WHILE field # NIL DO
    IF field.isNull THEN field.len := NullData
    ELSIF field IS StringField THEN field.len := NTS
    ELSIF field IS FileField THEN field.len := ODBC.DataAtExec
    ELSE field.len := 0
    END ;
    NextField(field)
  END ;
  s.ret := ODBC.SQLExecute(s.stmt);
  WHILE s.ret = NeedData DO (* for all unbound FileFields *)
    FirstField(s.params, field);
    s.ret := ODBC.SQLParamData(s.stmt, dummy);
    IF s.ret = NeedData THEN
      WHILE (field # NIL) & ~(field IS FileField) DO NextField(field) END ;
      IF field = NIL THEN HALT(45)
```

```
        ELSE (* f is next FileField *)
          Files.Set(r, field(FileField).f, 0);
          REPEAT
            Files.ReadBytes(r, buffer, BlockSize);
            ret := ODBC.SQLPutData(s.stmt, S.ADR(buffer), BlockSize - r.res);
            IF ret < Success THEN s.ret := ret; RETURN END
          UNTIL r.eof OR (ret # Success);
        END ;
        NextField(field)
      END
    END ;
    FirstField(s.params, field);
    ...
END Execute;
```

4.3 WHERE-Klauseln mit Nullwerten

Eine Besonderheit hat man bei WHERE-Klauseln mit Nullwerten zu beachten. Vor der Ausführung folgender SQL-Abfrage muß man sich fragen, ob die Parameter, die an die Platzhalter gebunden werden, Nullwerte sind oder nicht. Wenn die Parameter Nullwerte sind, muß man die SQL-Abfrage umformulieren, da ansonsten keine Datensätze gefunden werden. Dieselbe Problematik gibt es in Embedded SQL.

`SELECT * FROM Friends WHERE (firstName = ?) AND (lastName = ?)`

muß man umformen in

`SELECT * FROM Friends WHERE (firstName IS NULL) AND (lastName = ?)`,

`SELECT * FROM Friends WHERE (firstName = ?) AND (lastName IS NULL)`

oder

`SELECT * FROM Friends WHERE (firstName IS NULL) AND (lastName IS NULL)`,

je nachdem, ob der erste Parameter (an erstes Fragezeichen gebunden) einen Nullwert enthält, der zweite Parameter oder beide Parameter Nullwerte enthalten.

4.4 Finalisierung

Zum Schließen der Verbindungen zu Datenbanken wird der Mechanismus der Finalisierung verwendet (siehe [Tem94]). Es wird für die Anweisungs- und Verbindungsobjekte jeweils eine Prozedur registriert, die aufgerufen wird, wenn der Garbage-Collector den Speicher zur Wiederverwendung einsammelt. In der Abbildung 2.3 sieht man, daß jede Anweisung einen Verweis auf die darunterliegende Verbindung hat, Verbindungsobjekte werden also sicher nicht vor den Anweisungsobjekten freigegeben. Es kann aber sein, daß Verbindungsobjekte nur mehr von Anweisungsobjekten referenziert werden, daß also beide (Verbindung und Anweisung) im selben Garbage-Collector-Lauf eingesammelt (und somit finalisiert) werden. In den

Finalisierungsprozeduren der Anweisungs- und Verbindungsobjekte werden ODBC-Funktionen aufgerufen, die Speicher innerhalb des Datenbanktreibers freigeben. Beim Aufruf dieser Funktionen muß man darauf achten, daß zuerst der Speicher für die Anweisungen freigegeben wird und dann erst der Speicher für die Verbindung. Wird der Speicher für Anweisungen nicht explizit freigegeben, wird er spätestens beim Freigeben des Speichers für die Verbindung (automatisch vom Datenbanktreiber) freigegeben. Da keine Reihenfolge der Finalisierung der Oberon-Objekte definiert ist und es möglich ist, daß die Finalisierungsprozedur der Anweisungen nach der der Verbindungen aufgerufen wird, darf man in diesem Fall (der Speicher für die Anweisungen ist ja implizit beim Freigeben des Speichers für die Verbindung schon freigegeben worden) die ODBC-Funktion zum Freigeben des Anweisungsspeicher nicht mehr aufrufen. Macht man das trotzdem, resultiert daraus eine Speicherschutzverletzung.

```
PROCEDURE FinalizeConnection (o: S.PTR);
  VAR ret: INTEGER; c: Connection;
BEGIN
  c := o(Connection);
  ret := ODBC.SQLDisconnect(c.dbc);
  ret := ODBC.SQLFreeConnect(c.dbc);
  c.finalized := TRUE
END FinalizeConnection;

PROCEDURE FinalizeStatement (o: S.PTR);
  VAR ret: ODBC.Retcode; s: Statement;
BEGIN
  s := o(Statement);
  IF ~s.conn.finalized THEN ret := ODBC.SQLFreeStmt(s.stmt, ODBC.Drop) END
END FinalizeStatement;
```

Kapitel 5

Erläuterungen zu Designentscheidungen

5.1 ODBC

Das Modul ODBC enthält Deklarationen von Typen, Konstanten und Prozeduren, die in der von Microsoft definierten ODBC-Schnittstelle enthalten sind. Im wesentlichen ist es eine 1:1-Abbildung der DLL ODBC32.DLL, die unter Windows den Treibermanager implementiert. Das Definitionsmodul wurde so gehalten, daß es möglichst gleich dem unter Oberon/F verfügbaren ist, um die Portabilität von Oberon-ODBC-Programmen nicht einzuschränken.

Es ist auch denkbar, diese Deklarationen in das Modul SQL zu integrieren, allerdings verliert man dadurch an Portabilität. Bei einer Portierung nach Oberon/F müßte man im Modul SQL dann wesentlich mehr Änderungen vornehmen.

Es ist auch durchaus einleuchtend, die DLL so direkt wie möglich abzubilden, da man dann auch ODBC-Programme, die in anderen Sprachen geschrieben sind und nur die ODBC-DLL verwenden, leicht nach Oberon portieren kann.

Erst in einem höheren Modul werden Prozeduren als Schnittstelle für Anwendungsprogrammierer definiert, die dann zum Beispiel bei Zeichenkettenparametern nicht mehr einen LONGINT-Parameter (Adresse des ersten Zeichens), sondern einen VAR ARRAY OF CHAR-Parameter erwarten.

5.2 Ausgliederung von SQLExt aus SQL

Die Module SQL und SQLExt bieten dem Anwendungsprogrammierer den Funktionsumfang von ODBC Extension Level 1, 2 und 3. Es ist durchaus denkbar die beiden Module in eines zusammenzufassen, allerdings wird nicht jedes Anwendungsprogramm die zusätzliche Funktionalität benötigen, würde dann aber trotzdem den gesamten Funktionsumfang laden.

Es soll durch diese Ausgliederung auch gezeigt werden, daß Erweiterungen hinzugefügt werden können, was insbesondere interessant ist, da bereits an einer Version 3.0 von ODBC gearbeitet wird.

5.3 Modellierung der Verbindungen und Anweisungen

Prinzipiell wird in [Moe93] folgende Abfolge von Fragen vorgeschlagen, um zu entscheiden, ob man etwas als konkrete Datenstruktur, abstrakte Datenstruktur, abstrakten Datentypen oder Klasse modelliert:

- Ist Datenabstraktion wichtig? Nein ⇒ konkrete Datenstruktur
- Gibt es mehrere Instanzen dieser Daten? Nein ⇒ abstrakte Datenstruktur
- Gibt es Varianten dieser Daten, die auf gleiche Weise behandelt werden sollten? Nein ⇒ abstrakter Datentyp, ja ⇒ Klasse

Wenn man diese Fragen für die Modellierung von Verbindungs- und Anweisungsstrukturen durchgeht, kommt man zu dem Ergebnis, daß abstrakte Datentypen wohl ausreichen und man nicht unbedingt Klassen braucht.

5.4 Methoden oder Prozeduren

Die Operationen, die man auf Verbindungen und Anweisungen anwenden kann, kann man als Methoden oder Prozeduren implementieren. Wenn man sie als Methoden implementiert, ist schon die Ausgabe von Browser.ShowDef recht übersichtlich gestaltet, man sieht sofort die Klassen mit deren Methoden. Ein Nachteil dabei ist allerdings, daß man alle Methoden zu einer Klasse in derselben Datei spezifizieren muß. Man kann also nicht in einem zusätzlichen Modul die erweiterte Funktionalität von ODBC Extension Level 2 und 3 anbieten. Man hätte dann zwei Arten von Operationen: die innerhalb des Grundmoduls definierten als Methoden und die ausserhalb definierten als Prozeduren. Diese Nichtgleichbehandlung und die Tatsache, daß Klassen konzeptuell nicht notwendig sind, gaben den Ausschlag, alle ODBC-Operationen als Prozeduren zu implementieren.

5.5 Iterator für Ergebniszeile

Um die Werte einer Ergebniszeile verarbeiten zu können, muß man über die Ergebniszeile iterieren können. Anfangs war daran gedacht, dafür einen Scanner mit ungefähr folgender Schnittstelle zu definieren:

```
Scanner = RECORD
  eor: BOOLEAN; (* end of row *)
  colIdx: INTEGER;
  sqlType: INTEGER;
  b: BOOLEAN;
  i: LONGINT;
  r: LONGREAL;
  str: ARRAY 256 OF CHAR;
  ts: RECORD year, month, day, hour, minute, second: INTEGER;
      fraction: LONGINT
    END ;
  f: Files.File
END ;

PROCEDURE OpenScanner (stat: Statement; VAR s: Scanner; colIdx: INTEGER);
PROCEDURE Scan (VAR s: Scanner);
```

Nach dem Öffnen eines Scanners über der Spalte mit der Nummer colIdx kann man den Scanner mit Scan vorrücken, bis das Feld eor (end of row) TRUE ist. Abhängig vom Typ der Spalte (sqlType) steht der eingelesene Wert in einem der Datenfelder.

Da aber mitunter wahlweiser Zugriff, beziehungsweise Zugriff über den Namen einer Spalte in der Ergebnismenge erwünscht ist, wurde der Ansatz der ODBC-Anbindung für Oberon System 3 von Markus Dätwyler verwendet:

```
PROCEDURE FindField (r: Row; name: ARRAY OF CHAR; VAR f: Field);
PROCEDURE FirstField (r: Row; VAR f: Field);
PROCEDURE NextField (VAR f: Field);
PROCEDURE PrevField (VAR f: Field);
```

Wie in der Prozedur ListAll in 2.4.6 gezeigt, kann man über die Felder einer Ergebniszeile iterieren. Mit Typtests und Typzusicherungen kann man auf die einzelnen Datenfelder der Recorderweiterungen zugreifen (siehe PutField in ODBCBench im Anhang B.2).

Kapitel 6

Geschwindigkeitsvergleich

6.1 ODBC mit verschiedenen Treibern

6.1.1 Access / Oracle

Die folgenden Tests wurden auf einem 100 MHz Pentium mit Windows for Workgroups 3.11, 16 MB Hauptspeicher, PCI-Bus, Adaptec SCSI-Kontroller und 1 MB SCSI-Festplatte, 20 MB permanenter Auslagerungsdatei, 512 kB Cache (Windows-Systemsteuerung, 386 erweitert, Virtueller Speicher) und 2048 SmartDrive-Cache durchgeführt.

Nach dem Laden von Windows und Oberon waren noch verfügbar:
21771 kB Speicher
72% System-Ressourcen

Nach dem Starten der Oracle-Datenbank waren nur mehr verfügbar:
10028 kB Speicher
70% System-Ressourcen

In den folgenden Tests wurden die Datenbanktreiber MS Access 2.0 (des ODBC SDKs) und Personal Oracle 7 miteinander verglichen. Die Verbindungszeichenketten zu den beiden Datenbanktreibern waren:
`"DSN=Access_sdk20;DBQ=C:\ODBCSDK\SMPLDATA\ACCESS\SAMPLE.MDB; FIL=RedISAM;"` und
`"DSN=TestDB;DBQ=2:;UID=SCOTT;PWD=TIGER"`.

Test 1: Verbinden mit Datenbank, viermal hintereinander die Testtabellen löschen und neu anlegen, Verbindung schließen und Garbage collector aufrufen. Da das Schließen der Verbindung implizit über den Mechanismus der Finalisierung (siehe Kapitel 4.4) passiert, ist es notwendig, einen Garbage collector-Durchlauf zu erzwingen. Die Zeit dafür beträgt allerdings nur Sekundenbruchteile und verfälscht das Ergebnis nicht.

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.DropTables
ODBCBench.CreateTables
ODBCBench.DropTables
ODBCBench.CreateTables
ODBCBench.DropTables
ODBCBench.CreateTables
ODBCBench.DropTables
ODBCBench.CreateTables
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 2: Verbinden mit Datenbank, in TestTabelle 1 100 Datensätze einfügen, Verbindung schließen und Garbage collector aufrufen.

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.InsertIntoBench1
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 3: Verbinden mit Datenbank, an TestTabelle 1 100 Datensätze anhängen, Verbindung schließen und Garbage collector aufrufen.

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.AppendToBench1
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 4: Verbinden mit Datenbank, in TestTabelle 2 100 Datensätze einfügen, Verbindung schließen und Garbage collector aufrufen.

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.InsertIntoBench2
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 5: Verbinden mit Datenbank, viermal „SELECT Bench1.val FROM Bench1, Bench2 WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ mit Zeichenkettenparameter (Wert: „%14%“) ausführen, Verbindung schließen und Garbage collector aufrufen (Testtabelle 1 enthält 200 Datensätze, Testtabelle 2 enthält 100 Datensätze)

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2
  WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%14%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2
  WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%14%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2
  WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%14%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2
  WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%14%“
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 6: Verbinden mit Datenbank, an TestTabelle 1 2000 Datensätze anhängen, Verbindung schließen und Garbage collector aufrufen.

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.InsertIntoBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
```

```
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.AppendToBench1
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 7: Verbinden mit Datenbank, viermal „SELECT Bench1.val FROM Bench1, Bench2 WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ mit Zeichenkettenparameter (Wert: „%1%“) ausführen, Verbindung schließen und Garbage collector aufrufen (Testtabelle 1 enthält 2000 Datensätze, Testtabelle 2 enthält 100 Datensätze)

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2
  WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%1%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2
  WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%1%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2
  WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%1%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2
  WHERE (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%1%“
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 8: Verbinden mit Datenbank, in TestTabelle 3 11 lange Datensätze einfügen und wieder auslesen (mit Verifikation), Verbindung schließen und Garbage collector aufrufen.

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.InsertLongData
```

```
ODBCBench.GetLongData
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 9: Verbinden mit Datenbank, „SELECT count(*) FROM Bench1, Bench1 Again“ (2000 Datensätze in Bench1),

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.Execute „SELECT count(*) FROM Bench1, Bench1 Again“
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Test 10: Verbinden mit Datenbank, Indizes erstellen, dann wie Test7

```
Batch.Start
System.Time
ODBCBench.Connect
ODBCBench.Execute „CREATE INDEX idx1 ON Bench1 (i)“
ODBCBench.Execute „CREATE INDEX idx2 ON Bench2 (i)“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2 WHERE
  (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%1%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2 WHERE
  (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%1%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2 WHERE
  (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%1%“
ODBCBench.ExecuteStringParam „SELECT Bench1.val FROM Bench1, Bench2 WHERE
  (Bench1.i = Bench2.i) AND (Bench1.firstname LIKE ?)“ „%1%“
ODBCBench.Disconnect
System.Collect
System.Time
~
```

Die in der folgenden Tabelle abgedruckten Ergebnisse sind die in Sekunden gemessenen Zeiten. Es fällt auf, daß der Datenbanktreiber von MS Access in den meisten Fällen ganz gut abschneidet, daß aber der Oracle-Datenbanktreiber bei Abfragen schneller Ergebnisse liefert (auch schon ohne Indizes).

Test-Nummer	Access20 (SDK)	Personal Oracle 7
1, Löschen und Anlegen von Tabellen	18	15
2, Einfügen von Datensätzen in Tabelle 1	10	22
3, Anhängen von Datensätzen in Tabelle 1	10	14
4, Einfügen von Datensätzen in Tabelle 2	10	23
5, Select mit Join	3	3
6, Anhängen von Datensätzen in Tabelle 1	200	256
7, Select mit Join und Zeichenkettenparameter	57	4
8, Einfügen von langen Daten in Tabelle 3	32	
9, Select mit Join, große Ergebnistabelle	97	45
10, wie 7 aber mit Indizes	6	7

Tabelle 6.1: Gemessene Zeiten (in Sekunden) für die einzelnen Tests

Kapitel 7

Portierung auf andere Oberon-Systeme

Die Portierung der Module ODBC, SQL und SQLExt auf andere Oberon-Systeme erwies sich, dank der gewählten Modularisierung, als sehr einfach und bedurfte nur weniger einfacher Änderungen. Das Modul EmbeddedSQL benötigt die vollständige Referenzinformation, wie es das Modul Ref unter dem Oberon System V4 anbietet. Um also das Modul EmbeddedSQL zu portieren, müßte man zuerst auf dem entsprechenden Zielsystem den Zugriff auf die vollständige Referenzinformation implementieren.

7.1 Oberon System 3

Die Portierung der Module ODBC, SQL und SQLExt beschränkte sich auf die Neuübersetzung der Module. Auch Testprogramme, die für die Ein- und Ausgabe nur die Module In und Out verwenden, sind einfach zu übernehmen. Mehr Aufwand wäre die Portierung der Anwendungsprogramme DBBrowser und DataEditor, da man die Benutzerschnittstelle mit Gadgets anstatt mit Dialogen gestalten müßte.

7.2 Oberon/F

Unter Oberon/F existiert ein Modul namens ODBC32, das die Abbildung der ODBC32.DLL darstellt. Wie schon in Kapitel 5.1 erwähnt, wurde die Schnittstelle des selber implementierten Moduls ODBC so definiert, daß sie dem unter Oberon/F gleicht. Deswegen mußten für die Portierung nach Oberon/F in den Modulen SQL und SQLExt nur kleine Änderungen vorgenommen werden.

Mit Hilfe von VersionElems [1] kann man zwischen den Versionen für Oberon/F

[1] VersionElems sind eine spezielle Art von Textelementen unter dem Oberon System V4 zur Verwaltung mehrerer Versionen eines Programmes innerhalb ein und derselben Datei. Eine Beschreibung findet man in [MöK96].

und System 3/4 umschalten.

Unter Oberon/F sind die Zeichenkettenparameter der Prozeduren des Moduls ODBC als vom Typ StringPtr, unter Oberon System V4 als LONGINT (Adresse der Zeichenkette) deklariert. Das bedingt, daß die Berechnung der Adresse (SYSTEM.ADR(string)) unter Oberon/F entfällt.

Weil die Behandlung langer Zeichenkettendaten und langer binärer Daten dateibasiert ist, mußte im Grundmodul SQL auch das Modul Files importiert werden. Die Schnittstelle des Moduls Files unter Oberon/F ist zwar anders als unter Oberon System V4, eine Übersetzung der einzelnen Prozeduraufrufe erfolgte aber ohne Probleme.

Weiters ist die Schnittstelle des Moduls Kernel, das für die Finalisierung der Objekte verwendet wird, etwas anders. Da aber Finalisierung von Objekten und Terminierung von Modulen unter Oberon/F verfügbar ist, muß man für die Portierung lediglich die unter Oberon/F dafür zuständigen Prozeduren kennen.

Anhang A

Definitionsmodule

A.1 ODBC

```
DEFINITION ODBC;

CONST
  Char = 1; CChar = Char; (* numeric values of predefined ODBC data types *)
  OChar = Char; Numeric = 2; Decimal = 3; Integer = 4; CLong = Integer;
  SmallInt = 5; CShort = SmallInt; Float = 6; Real = 7; CFloat = Real; OReal = Real;
  Double = 8; CDouble = Double; OLReal = CDouble; Date= 9; CDate = Date;
  Time = 10; CTime = Time; TimeStamp = 11; CTimeStamp = TimeStamp;
  VarChar = 12; LongVarChar = -1; Binary = -2; CBinary = Binary;
  VarBinary = -3; LongVarBinary = -4; BigInt = -5; TinyInt = -6;
  CTinyInt = TinyInt; Bit = -7; CBit = Bit; CDefault = 99; CBookmark = -18;
  CSShort = -15; CSLong = -16; CUShort = -17; CULong = -18; CSTinyInt = -26;
  CUTinyInt = -28; SignedOffset = -20; UnsignedOffset = -22;

  InvalidHandle = -2; Error = -1; Success = 0; SuccessWithInfo = 1; (* error codes *)
  StillExecuting = 2; NeedData = 99; NoDataFound = 100; NoTotal = -4;

  NTS = -3; NullData = -1; (* values for parameter length field *)

  NullDBC = 0; NullENV = 0; NullSTMT = 0;

  ParamInput = 1; ParamInOut = 2; ParamResultCol = 3; (* kinds of parameters *)
  ParamOutput = 4;

  FetchNext = 1; FetchFirst = 2; FetchLast = 3; FetchPrior = 4; (* cursor directions *)
  FetchAbsolute = 5; FetchRelative = 6; FetchBookmark = 8;

  MaxMessageLength = 512; MaxOptionStringLen = 256;

  APIAllFunctions = 0; AllTypes = 0;
```

```
  Commit = 0; RollBack = 1; (* values for SQLTransact(..., type) *)

  Close = 0; Drop = 1; Unbind = 2; ResetParams = 3; (* values for SQLFreeStm(..., option) *)

  NoNulls = 0; Nullable = 1; NullableUnknown = 2;

  DriverNoPrompt = 0; DriverComplete = 1; (* values for SQLDriverConnect(..., completion) *)
  DriverPrompt = 2; DriverCompleteRequired = 3;

  DataAtExec = -2; DefaultParam = -5; Ignore = -6; LenDataAtExecOffset = -100;

  LockNoChange = 0; LockExclusive = 1; LockUnlock = 2;

  ConcurReadOnly = 1; ConcurLock = 2; ConcurRowVer = 3; ConcurValues = 4;

  ScopeCurRow = 0; ScopeTransaction = 1; ScopeSession = 2;

  Unsearchable = 0; LikeOnly = 1; AllExceptLike = 2; Searchable = 3;

  Position = 0; Refresh = 1; Update = 2; Delete = 3; Add = 4;

  ColumnCount = 0; ColumnName = 1; ColumnType = 2; ColumnLength = 3;
  ColumnPrecision = 4; ColumnScale = 5; ColumnDisplaySize = 6; ColumnNullable = 7;
  ColumnUnsigned = 8; ColumnMoney = 9; ColumnUpdatable = 10;
  ColumnAutoIncrement = 11; ColumnCaseSensitive = 12; ColumnSearchable = 13;
  ColumnTypeName = 14; ColumnTableName = 15; ColumnOwnerName = 16;
  ColumnQualifierName = 17; ColumnLabel = 18; ColumnDriverStart = 1000;

  AttrReadOnly = 0; AttrWrite = 1; AttrReadWriteUnknown = 2;


TYPE
  Bookmark = LONGINT;
  DateRec = RECORD
    year, month, day: INTEGER
  END ;
  TimeRec = RECORD
    hour, minute, second: INTEGER
  END ;
  TimeStampRec = RECORD
    year, month, day, hour, minute, second: INTEGER;
    fraction: LONGINT
  END ;
  Ptr = LONGINT;
  ENV = LONGINT;
```

```
  DBC = LONGINT;
  STMT = LONGINT;
  Retcode = INTEGER;

VAR
  SQLAllocConnect-: PROCEDURE (henv: LONGINT;
    VAR hdbc: LONGINT): INTEGER;
  SQLAllocEnv-: PROCEDURE (VAR henv: LONGINT): INTEGER;
  SQLAllocStmt-: PROCEDURE (hdbc: LONGINT;
    VAR hstmt: LONGINT): INTEGER;
  SQLBindCol-: PROCEDURE (hstmt: LONGINT; colIdx,
    colType: INTEGER; buf, bufMaxLen: LONGINT;
    VAR bufActLen: LONGINT): INTEGER;
  SQLBindParameter-: PROCEDURE (hstmt: LONGINT; parIdx,
    parType, cType, sqlType: INTEGER; prec: LONGINT;
    scale: INTEGER; buf, bufMaxLen: LONGINT;
    VAR bufActLen: LONGINT): INTEGER;
  SQLBrowseConnect-: PROCEDURE (hdbc, connStrIn: LONGINT;
    connStrInLen: INTEGER; connStrOut: LONGINT;
    connStrOutMaxLen: INTEGER;
    VAR connStrOutActLen: INTEGER): INTEGER;
  SQLCancel-: PROCEDURE (hstmt: LONGINT): INTEGER;
  SQLColAttributes-: PROCEDURE (hstmt: LONGINT; colIdx,
    descType: INTEGER; desc: LONGINT; descMaxLen: INTEGER;
    VAR descActLen: INTEGER; VAR descFlag: LONGINT): INTEGER;
  SQLColumnPrivileges-: PROCEDURE (hstmt, tableQual: LONGINT;
    tableQualLen: INTEGER; tableOwner: LONGINT; tableOwnerLen: INTEGER;
    table: LONGINT; tableLen: INTEGER; col: LONGINT;
    colLen: INTEGER): INTEGER;
  SQLColumns-: PROCEDURE (hstmt, tableQual: LONGINT;
    tableQualLen: INTEGER; tableOwner: LONGINT; tableOwnerLen: INTEGER;
    table: LONGINT; tableLen: INTEGER; col: LONGINT;
    colLen: INTEGER): INTEGER;
  SQLConnect-: PROCEDURE (hdbc, DSN: LONGINT; DSNLen: INTEGER;
    UID: LONGINT; cbUID: INTEGER; AuthStr: LONGINT;
    cbAuthStr: INTEGER): INTEGER;
  SQLDataSources-: PROCEDURE (henv: LONGINT; direction: INTEGER;
    DSN: LONGINT; DSNMaxLen: INTEGER; VAR DSNActLen: INTEGER;
    description: LONGINT; descriptMaxLen: INTEGER;
    VAR descripActLen: INTEGER): INTEGER;
  SQLDescribeCol-: PROCEDURE (hstmt: LONGINT; colIdx: INTEGER;
    colName: LONGINT; colNameMaxLen: INTEGER; VAR colNameActLen,
    sqlType: INTEGER; VAR prec: LONGINT; VAR scale,
    nullable: INTEGER): INTEGER;
  SQLDescribeParam-: PROCEDURE (hstmt: LONGINT; parIdx: INTEGER;
    VAR sqlType: INTEGER; VAR prec: LONGINT; VAR scale,
```

```
    nullable: INTEGER): INTEGER;
  SQLDisconnect-: PROCEDURE (hdbc: LONGINT): INTEGER;
  SQLDriverConnect-: PROCEDURE (hdbc, hwnd, connStrIn: LONGINT;
    connStrInLen: INTEGER; connStrOut: LONGINT;
    connStrOutMaxLen: INTEGER; VAR connStrOutActLen: INTEGER;
    completion: INTEGER): INTEGER;
  SQLDrivers-: PROCEDURE (henv: LONGINT; direction: INTEGER;
    driver: LONGINT; driverMaxLen: INTEGER; VAR driverActLen: INTEGER;
    attributes: LONGINT; attribMaxLen: INTEGER;
    VAR attribActLen: INTEGER): INTEGER;
  SQLError-: PROCEDURE (henv, hdbc, hstmt, sqlState: LONGINT;
    VAR nativeErr: LONGINT; errMsg: LONGINT; errMsgMaxLen: INTEGER;
    VAR errMsgActLen: INTEGER): INTEGER;
  SQLExecDirect-: PROCEDURE (stmt, sqlStr, sqlStrLen: LONGINT): INTEGER;
  SQLExecute-: PROCEDURE (hstmt: LONGINT): INTEGER;
  SQLExtendedFetch-: PROCEDURE (stmt: LONGINT; fetchType: INTEGER;
    rowIn: LONGINT; VAR rowOut: LONGINT;
    VAR status: ARRAY OF INTEGER): INTEGER;
  SQLFetch-: PROCEDURE (hstmt: LONGINT): INTEGER;
  SQLForeignKeys-: PROCEDURE (hstmt, pkTabQual: LONGINT;
    pkTabQualLen: INTEGER; pkTabOwner: LONGINT; pkTabOwnerLen: INTEGER;
    pkTable: LONGINT; pkTabLen: INTEGER; fkTabQual: LONGINT;
    fkTabQualLen: INTEGER; fkTabOwner: LONGINT; fkTabOwnerLen: INTEGER;
    fkTable: LONGINT; fkTabLen: INTEGER): INTEGER;
  SQLFreeConnect-: PROCEDURE (hdbc: LONGINT): INTEGER;
  SQLFreeEnv-: PROCEDURE (henv: LONGINT): INTEGER;
  SQLFreeStmt-: PROCEDURE (hstmt: LONGINT; option: INTEGER): INTEGER;
  SQLGetConnectOption-: PROCEDURE (hdbc: LONGINT; option: INTEGER;
    bufAdr: LONGINT): INTEGER;
  SQLGetCursorName-: PROCEDURE (hstmt, cursor: LONGINT;
    cursorMaxLen: INTEGER; VAR cursorActLen: INTEGER): INTEGER;
  SQLGetData-: PROCEDURE (hstmt: LONGINT; colIdx, type: INTEGER; buf,
    bufMaxLen: LONGINT; VAR bufActLen: LONGINT): INTEGER;
  SQLGetFunctions-: PROCEDURE (hdbc: LONGINT; functionNr: INTEGER;
    VAR exists: INTEGER): INTEGER;
  SQLGetInfo-: PROCEDURE (hdbc: LONGINT; infoType: INTEGER;
    buf: LONGINT; bufMaxLen: INTEGER;
    VAR bufActLen: INTEGER): INTEGER;
  SQLGetStmtOption-: PROCEDURE (hstmt: LONGINT; option: INTEGER;
    buf: LONGINT): INTEGER;
  SQLGetTypeInfo-: PROCEDURE (hstmt: LONGINT;
    sqlType: INTEGER): INTEGER;
  SQLMoreResults-: PROCEDURE (hstmt: LONGINT): INTEGER;
  SQLNativeSql-: PROCEDURE (hdbc, stat, statActLen, nativeStat,
    nativeStatMaxLen: LONGINT;
    VAR nativeStatActLen: LONGINT): INTEGER;
```

```
SQLNumParams-: PROCEDURE (hstmt: LONGINT;
  VAR params: INTEGER): INTEGER;
SQLNumResultCols-: PROCEDURE (hstmt: LONGINT;
  VAR cols: INTEGER): INTEGER;
SQLParamData-: PROCEDURE (hstmt, adr: LONGINT): INTEGER;
SQLParamOptions-: PROCEDURE (stmt, row: LONGINT;
  VAR num: LONGINT): INTEGER;
SQLPrepare-: PROCEDURE (hstmt, sqlStr, sqlStrLen: LONGINT): INTEGER;
SQLPrimaryKeys-: PROCEDURE (hstmt, tableQual: LONGINT;
  tableQualLen: INTEGER; tableOwner: LONGINT; tableOwnerLen: INTEGER;
  table: LONGINT;
  tableLen: INTEGER): INTEGER;
SQLProcedureColumns-: PROCEDURE (hstmt, procQual: LONGINT;
  procQualLen: INTEGER; procOwnerPat: LONGINT; procOwnerLen: INTEGER;
  procPat: LONGINT; procLen: INTEGER; colPat: LONGINT;
  colLen: INTEGER): INTEGER;
SQLProcedures-: PROCEDURE (hstmt, procQual: LONGINT;
  procQualLen: INTEGER; procOwnerPat: LONGINT; procOwnerLen: INTEGER;
  procPat: LONGINT;
  procLen: INTEGER): INTEGER;
SQLPutData-: PROCEDURE (hstmt, buf, bufLen: LONGINT): INTEGER;
SQLRowCount-: PROCEDURE (hstmt: LONGINT;
  VAR rows: LONGINT): INTEGER;
SQLSetConnectOption-: PROCEDURE (hdbc: LONGINT; option: INTEGER;
  val: LONGINT): INTEGER;
SQLSetCursorName-: PROCEDURE (hstmt, cursorName: LONGINT;
  cursorNameActLen: INTEGER): INTEGER;
SQLSetPos-: PROCEDURE (hstmt: LONGINT; rowIdx, option,
  lock: INTEGER): INTEGER;
SQLSetScrollOptions-: PROCEDURE (hstmt: LONGINT; concurrency: INTEGER;
  rowsInKeyset: LONGINT; rowsInRowset: INTEGER): INTEGER;
SQLSetStmtOption-: PROCEDURE (hstmt: LONGINT; option: INTEGER;
  val: LONGINT): INTEGER;
SQLSpecialColumns-: PROCEDURE (hstmt: LONGINT; colType: INTEGER;
  tableQual: LONGINT; tableQualLen: INTEGER; tableOwner: LONGINT;
  tableOwnerLen: INTEGER; table: LONGINT; tableLen, scope,
  nullable: INTEGER): INTEGER;
SQLStatistics-: PROCEDURE (hstmt, tableQual: LONGINT;
  tableQualLen: INTEGER; tableOwnerPat: LONGINT; tableOwnerLen: INTEGER;
  table: LONGINT; tableLen, indexType, accuracy: INTEGER): INTEGER;
SQLTablePrivileges-: PROCEDURE (hstmt, tableQual: LONGINT;
  tableQualLen: INTEGER; tableOwnerPat: LONGINT; tableOwnerLen: INTEGER;
  table: LONGINT; tableLen: INTEGER): INTEGER;
SQLTables-: PROCEDURE (hstmt, tableQual: LONGINT; tableQualLen: INTEGER;
  tableOwnerPat: LONGINT; tableOwnerLen: INTEGER; tablePat: LONGINT;
  tableLen: INTEGER; typesList: LONGINT; typesListLen: INTEGER): INTEGER;
```

```
  SQLTransact-: PROCEDURE (henv, hdbc: LONGINT; type: INTEGER): INTEGER;

END ODBC.
```

A.2 EmbeddedSQL

```
DEFINITION EmbeddedSQL;

CONST
  InvHandle = -2; Error = -1; Success = 0; (* return codes *)
  SuccessWithInfo = 1; NoDataFound = 100;
  showErrors = 1; haltOnErrors = 2; verboseErrorMsg = 3; (* options *)

TYPE
  Connection = POINTER TO ConnectionD;
  ConnectionD = RECORD
    ret: INTEGER; (* return code of the last operation *)
    dbc-: LONGINT; (* connection handle *)
    PROCEDURE (c: Connection) Prepare (sqlString: ARRAY OF CHAR): Statement;
    PROCEDURE (c: Connection) Error (str: ARRAY OF CHAR);
    PROCEDURE (c: Connection) Commit;
    PROCEDURE (c: Connection) Rollback;
    PROCEDURE (c: Connection) GetIntInfo (infoNr: INTEGER; VAR info: LONGINT);
    PROCEDURE (c: Connection) GetStrInfo (infoNr: INTEGER;
      VAR info: ARRAY OF CHAR);
    PROCEDURE (c: Connection) GetIntOption (option: INTEGER;
      VAR info: LONGINT);
    PROCEDURE (c: Connection) SetIntOption (option: INTEGER; info: LONGINT);
    PROCEDURE (c: Connection) GetStrOption (option: INTEGER;
      VAR info: ARRAY OF CHAR);
    PROCEDURE (c: Connection) SetStrOption (option: INTEGER;
      info: ARRAY OF CHAR);
  END ;

  Statement = POINTER TO StatementD;
  StatementD = RECORD
    ret: INTEGER; (* return code of the last operation *)
    stmt-: LONGINT; (* statement handle *)
    conn-: Connection; (* the connection on which the statement is executed *)
    PROCEDURE (s: Statement) Execute;
    PROCEDURE (s: Statement) IsNull (varName: ARRAY OF CHAR): BOOLEAN;
    PROCEDURE (s: Statement) SetNull (varName: ARRAY OF CHAR);
    PROCEDURE (s: Statement) Fetch (): BOOLEAN;
    PROCEDURE (s: Statement) Error (str: ARRAY OF CHAR);
    PROCEDURE (s: Statement) RowCount (): LONGINT;
```

```
    PROCEDURE (s: Statement) GetOption (option: INTEGER; VAR info: LONGINT);
    PROCEDURE (s: Statement) SetOption (option: INTEGER; info: LONGINT);
  END ;

VAR
  env-: ODBC.ENV; (* environment handle for the ODBC session *)
  options: SET;

PROCEDURE Open (source, user, passwd: ARRAY OF CHAR): Connection;
PROCEDURE OpenUI (connStrIn: ARRAY OF CHAR;
  VAR connStrOut: ARRAY OF CHAR): Connection;
PROCEDURE SetOption;
PROCEDURE SQLError (env, dbc, stmt: LONGINT; VAR sqlState: ARRAY OF CHAR;
  VAR nativeErr: LONGINT; VAR errMsg: ARRAY OF CHAR): INTEGER;
PROCEDURE ShowError (ret: INTEGER; str: ARRAY OF CHAR; env, dbc,
  stmt: LONGINT);

END EmbeddedSQL.
```

A.3 SQL

```
DEFINITION SQL;

IMPORT Files;

CONST
  InvHandle = -2; Error = -1; Success = 0; (* return codes *)
  SuccessWithInfo = 1; NeedData = 99; NoDataFound = 100;

  showErrors = 1; haltOnErrors = 2; verboseErrorMsg = 3; (* options *)

  InParam = 1; InOutParam = 2; OutParam = 4; (* parameter types  *)

  NTS = -3; NullData = -1; (* parameter values *)

  OberonChar = 1; OberonShortInt = -6; OberonBoolean = -7;
  OberonInteger = 5; OberonLongInt = 4; OberonReal = 7;
  OberonLongReal = 8; OberonBinary = -2; OberonFile = -4;
  OberonDate = 9; OberonTime = 10; OberonTimeStamp = 11;
  SQLBit = -7; SQLTinyInt = -6; SQLBigInt = -5;
  SQLLVarBin = -4; SQLVarBin = -3; SQLBin = -2;
  SQLLVarChar = -1; SQLChar = 1; SQLNumeric = 2;
  SQLDecimal = 3; SQLInteger = 4; SQLSmallInt = 5;
  SQLFloat = 6; SQLReal = 7; SQLDouble = 8;
```

```
  SQLDate = 9; SQLTime = 10; SQLTimeStamp = 11;
  SQLVarChar = 12;

TYPE
  Type = RECORD
    sqlType: INTEGER;
    prec: LONGINT; (* maximum number of significant digits *)
    scale: INTEGER (* -"- right to period *)
  END ;
  Field = POINTER TO FieldDesc;
  FieldDesc = RECORD
    name: ARRAY 32 OF CHAR; (* of the column or parameter *)
    len: LONGINT;
    type: Type;
    isNull, nullable: BOOLEAN
  END ;
  IntField = POINTER TO IntFieldDesc;
  IntFieldDesc = RECORD (FieldDesc) i: LONGINT END ;
  FileField = POINTER TO FileFieldDesc;
  FileFieldDesc = RECORD (FieldDesc) f: Files.File END ;
  StringField = POINTER TO StringFieldDesc;
  StringFieldDesc = RECORD (FieldDesc) str: ARRAY 256 OF CHAR END ;
  BinaryField = POINTER TO BinaryFieldDesc;
  BinaryFieldDesc = RECORD (FieldDesc) b: ARRAY 256 OF CHAR END ;
  BooleanField = POINTER TO BooleanFieldDesc;
  BooleanFieldDesc = RECORD (FieldDesc) b: BOOLEAN END ;
  DateField = POINTER TO DateFieldDesc;
  DateFieldDesc = RECORD (FieldDesc) year, month, day: INTEGER END ;
  TimeField = POINTER TO TimeFieldDesc;
  TimeFieldDesc = RECORD (FieldDesc) hour, minute, second: INTEGER END ;
  TimeStampField = POINTER TO TimeStampFieldDesc;
  TimeStampFieldDesc = RECORD (FieldDesc)
    year, month, day, hour, minute, second: INTEGER;
    fraction: LONGINT
  END ;
  RealField = POINTER TO RealFieldDesc;
  RealFieldDesc = RECORD (FieldDesc) r: LONGREAL END ;
  ParamDesc = RECORD
    type, mapType: Type;
    inOut: INTEGER;
    name: ARRAY 32 OF CHAR
  END ;
  Row = POINTER TO RowDesc;
  RowDesc = RECORD cols: INTEGER END ;
  Connection = POINTER TO ConnectionD;
  ConnectionD = RECORD
```

```
    ret: INTEGER; (* error code of last operation *)
    dbc-: LONGINT (* ODBC connection handle *)
  END ;
  Statement = POINTER TO StatementD;
  StatementD = RECORD
    ret: INTEGER; (* error code of last operation *)
    stmt-: LONGINT; (* ODBC statement handle *)
    conn-: Connection; (* associated connection *)
    results, params: Row (* row of results and parameters *)
  END ;

  EnumDataSourcesCB = PROCEDURE (dataSource, description: ARRAY OF CHAR;
    VAR continue: BOOLEAN);
  EnumDataTypesCB = PROCEDURE (typeName: ARRAY OF CHAR;
    dataType: INTEGER; prec: LONGINT; literalPrefix, literalSuffix,
    createPars: ARRAY OF CHAR; nullable: INTEGER; caseSensitive: BOOLEAN;
    searchable, unsigned: INTEGER; money: BOOLEAN; autoInc: INTEGER;
    localTypeName: ARRAY OF CHAR; minScale, maxScale: INTEGER;
    VAR continue: BOOLEAN);
  EnumDriversCB = PROCEDURE (driver, attributes: ARRAY OF CHAR;
    VAR continue: BOOLEAN);

VAR
  env-: LONGINT; (* environment handle for the ODBC session *)
  nullString-: ARRAY 2 OF CHAR;
  options: SET;

PROCEDURE Open (source, user, passwd: ARRAY OF CHAR): Connection; (* opens a
  connection to the specified database source with given user identification and password. *)
PROCEDURE OpenUI (connStrIn: ARRAY OF CHAR;
  VAR connStrOut: ARRAY OF CHAR): Connection; (* opens a connection as specified
  in connStrIn with possibly further user-interaction via dialog boxes. connStrOut
  is the full connection string that can be used the next time to establish the connection
  again. *)
PROCEDURE ShowError (ret: INTEGER; str: ARRAY OF CHAR; env, dbc,
  stmt: LONGINT);
PROCEDURE SetOption; (* sets or resets the specified option, valid options are 1
  (showErrors), 2 (haltOnErrors), and 3 (verboseErrorMsg). By specifying a negative value
  the option is reset. *)

PROCEDURE PrepareStatement (c: Connection;
  sqlString: ARRAY OF CHAR): Statement; (* prepares an SQL statement for execution.
  If the sqlString returns any results (e.g. a SELECT statement) the member variable results
  of the statement will point to a row containing a field for each column of the result set,
  otherwise the member variable results of the statement is NIL. After PrepareStatement the
  record field len of each element of results contains the maximum number of characters needed
```

to represent the corresponding data. The value in this record field len will only be valid until a call to Execute, so if you need this information you must check it between the calls to PrepareStatement and to Execute. *)
PROCEDURE **ErrC** (str: ARRAY OF CHAR; c: Connection);
PROCEDURE **Tables** (c: Connection): Statement; (* *returns a prepared statement holding a list of all table names in the specified data source.* *)
PROCEDURE **Commit** (c: Connection); (* *commits all statements associated with the specified connection c (if supported by the database).* *)
PROCEDURE **Rollback** (c: Connection); (* *rolls back all statements executed on the connection c since the last Commit (if supported by the database).* *)
PROCEDURE **Map** (c: Connection; type: Type; VAR mapType: Type): BOOLEAN; (* *maps the desired sqlType with the specified precision and scale to the data type that is supported by the connection (e.g. BIGINT may be mapped to CHAR(20), sqlType=−5, mapType=1, mapPrec=20).* *)
PROCEDURE **GetTypeName** (c: Connection; sqlType: INTEGER; VAR typeName, createParams: ARRAY OF CHAR); (* *returns the name of the SQL data type as it is used in the specified data source connection (e.g. BIGINT may be mapped to CHAR(20), sqlType=−5, mapType=1, mapPrec=20).* *)
PROCEDURE **GetConnOption** (c: Connection; option: INTEGER; info: Field); (* *returns the current setting of the specified option. Valid values for option and info are listed in the section C.2. The info field must be either an IntField or a StringField depending on the value of option.* *)
PROCEDURE **SetConnOption** (c: Connection; option: INTEGER; info: Field); (* *sets options that govern aspects of the connection. Valid values for option and field are listed in the section C.2. The info field must be either an IntField or a StringField depending on the value of option.* *)
PROCEDURE **GetInfo** (c: Connection; infoNr: INTEGER; info: Field); (* *returns general information about the driver and data source associated with the connection. Valid values for infoNr and info are listed in the section C.3. The info field must be either an IntField or a StringField depending on the value of infoNr.* *)
PROCEDURE **NewConnection** (): Connection; (* *returns a newly initialized connection which is not yet connected with a data source.* *)

PROCEDURE **BindParameters** (s: Statement; types: ARRAY OF ParamDesc; numParams: INTEGER); (* *binds the parameter fields to the statement s. The array of parameters contains a description of each parameter: types[i].type determines the type to be used in Oberon (i.e. what sort of Field (IntField, FileField, ...) should be added to params). types[i].mapType determines the type used in the data source (SQLInteger, SQLLVarBin, ...). types[i].inOut determines for which operation the parameter is used: data transfer from the applcation to the database* => *InParam (parameter used in an INSERT statement or in a stored procedure), to data transfer from the database to the application* => *OutParam (parameter is used in a stored procedure to get data), or bidirectional data transfer* => *InOutParam (parameter is used in a stored procedure for input and output). types[i].name can be used to name the parameter (i.e. you can find the corresponding Field in the member variable params of the statement with the procedure FindField).* *)
PROCEDURE **Execute** (s: Statement); (* *executes the previously prepared statement. If the*

statement delivers any data it can be retrieved after Execute with the procedure Fetch. IMPORTANT: a statement which does not need parameters and does not return any data (e.g. "DROP TABLE MyTable") will take effect even if you do not call Execute for this statement. *)

PROCEDURE **RowCount** (s: Statement; VAR rows: LONGINT); (* *returns the number of rows affected by the execution of s. This is not the number of rows which are delivered by a SELECT statement but rather the number of rows affected by an UPDATE or DELETE statement. If you want the number of rows in the result set of a SELECT statement use "SELECT COUNT(*) ..." or increment a counter variable after each successful call to Fetch.* *)

PROCEDURE **Fetch** (s: Statement); (* *fetches the next result row for statement s (statement s must, of course, have been executed before calling Fetch* *)

PROCEDURE **ErrS** (str: ARRAY OF CHAR; s: Statement);

PROCEDURE **GetStatementOption** (s: Statement; option: INTEGER; info: Field); (* *returns the current setting of the specified statement option. Valid values for option and info are listed in the section C.1. The info field must be either an IntField or a StringField depending on the value of option.* *)

PROCEDURE **SetStatementOption** (s: Statement; option: INTEGER; info: Field); (* *sets options related with the statement. To set options associated with the connection, use SetConnOption. Valid values for option and info are listed in the section C.1. The info field must be either an IntField or a StringField depending on the value of option.* *)

PROCEDURE **NewStatement** (c: Connection): Statement; (* *returns a newly initialized statement which is associated with the connection c.* *)

PROCEDURE **GenerateResultRow** (s: Statement); (* *creates the result row with the necessary fields for the prepared statement s.* *)

PROCEDURE **FirstField** (r: Row; VAR f: Field); (* *sets f to the first field in row r . If row = NIL, f is NIL too.* *)

PROCEDURE **NextField** (VAR f: Field); (* *sets f to the next field in the row containing f, NIL if there are no more fields.* *)

PROCEDURE **PrevField** (VAR f: Field); (* *sets f to the previous field in the row containing f, NIL if there is no previous field.* *)

PROCEDURE **FindField** (r: Row; name: ARRAY OF CHAR; VAR f: Field); (* *sets f to the field named name in row r.* *)

PROCEDURE **EnumDataSources** (cb: EnumDataSourcesCB); (* *enumerates all data sources, by setting the parameter continue of the callback procedure to FALSE you can end the enumeration process.* *)

PROCEDURE **EnumDataTypes** (c: Connection; cb: EnumDataTypesCB; sqlType: INTEGER); (* *enumerates all data types supported by the database for the specified sqlType (may be 0 in order to enumerate all data types), by setting the parameter continue of the callback procedure to FALSE you can end the enumeration process.* *)

PROCEDURE **EnumDrivers** (cb: EnumDriversCB); (* *enumerates all ODBC database drivers, by setting the parameter continue of the callback procedure to FALSE you can end the enumeration process.* *)

PROCEDURE **SQLError** (env, dbc, stmt: LONGINT; VAR sqlState: ARRAY OF CHAR; VAR nativeErr: LONGINT; VAR errMsg: ARRAY OF CHAR): INTEGER;

END SQL.

A.4 SQLExt

DEFINITION SQLExt;

IMPORT SQL;

PROCEDURE AppendType (c: SQL.Connection; VAR str: ARRAY OF CHAR;
type: SQL.Type); (* *appends the name of the Oberon data type, which is mapped to its corresponding DB type, and its creation parameters to str.* *)
PROCEDURE BrowseConnect (connStrIn: ARRAY OF CHAR;
VAR connStrOut: ARRAY OF CHAR); (* *supports an iterative method of discovering and enumerating the attributes and attribute values required to connect to a data source. Each call of BrowseConnect returns successive levels of attributes and attribute values. When all levels have been enumerated (c.ret is no longer SQL.NeedData), a connection to the data source is completed and a connection string is returned which can be used as input parameter to SQL.OpenUI.* *)
PROCEDURE Cancel (s: SQL.Statement); (* *cancels the processing on the statement. A function may be running asynchronously, a function may need data.* *)
PROCEDURE ColumnPrivileges (c: SQL.Connection; tableQual, tableOwner,
table, colPat: ARRAY OF CHAR): SQL.Statement; (* *returns a prepared statement holding the list of coluns and associated privileges for the specified table.* *)
PROCEDURE Columns (c: SQL.Connection; tableQual, tableOwnerPat, tablePat,
colPat: ARRAY OF CHAR): SQL.Statement; (* *returns a prepared statement holding the list of column names in the specified tables.* *)
PROCEDURE DataTypes (c: SQL.Connection; sqlType: INTEGER): SQL.Statement;
(* *returns a prepared statement holding the list of data types supported by the database for the specified sqlType (may be 0 inorder to enumerate all data types).* *)
PROCEDURE DescribeParam (s: SQL.Statement; parIdx: INTEGER;
VAR sqlType: INTEGER; VAR prec: LONGINT; VAR scale, nullable: INTEGER); (* *returns the description of the specified parameter. parIdx* $\geq$ *1. nullable can be 0 for the parameter does not allow NULL values, 1 for the parameter allows NULL values, and 2 for the driver cannot determine if the parameter allows NULL values.* *)
PROCEDURE ForeignKeys (c: SQL.Connection; pkTableQual, pkTableOwner,
pkTable, fkTableQual, fkTableOwner, fkTable: ARRAY OF CHAR):
SQL.Statement; (* *returns a prepared statement holding the list of foreign keys in the specified table (columns in the specified table that refer to primary keys in other tables) or the list of foreign keys in other tables that refer to the primary key in the specified table.* *)
PROCEDURE FunctionImplemented (c: SQL.Connection;
functionNr: INTEGER): BOOLEAN; (* *returns TRUE when the driver (Driver Manager or driver itself) supports the specified ODBC function.* *)
PROCEDURE GetColAttrib (s: SQL.Statement; colIdx, type: INTEGER;
VAR val: LONGINT); (* *returns descriptor information for the specified column in the result set, it cannot be used to return information about a bookmark column (colIdx=0).*

The descriptor information is returned as a 32-bit descriptor dependent value or an integer value in val. *)

PROCEDURE GetColAttribStr (s: SQL.Statement; colIdx, type: INTEGER;
VAR val: ARRAY OF CHAR); (* *returns descriptor information for the specified column in the result set, it cannot be used to return information about a bookmark column (colIdx=0). The descriptor information is returned as a character string in val.* *)

PROCEDURE GetCursorName (s: SQL.Statement;
VAR cursorName: ARRAY OF CHAR); (* *returns the cursor name associated with the statement. The only ODBC SQL statements that use a cursor name are position update and delete (for example "UPDATE myTable ... WHERE CURRENT OF cursorName"). If the application does not call SetCursorName to define a cursor name, on execution of a SELECT statement the driver generates a name that begins with the letters SQL_CUR and does not exceed* 18 *characters in length.* *)

PROCEDURE MoreResults (s: SQL.Statement); (* *determines whether there are more results available on the statement (should be a SELECT, UPDATE, INSERT, or DELETE statement) and, if so, initializes processing of those results* *)

PROCEDURE NativeSQLStatement (c: SQL.Connection;
sqlIn: ARRAY OF CHAR; VAR sqlOut: ARRAY OF CHAR); (* *returns the SQL string as translated by the driver.* *)

PROCEDURE NumParams (s: SQL.Statement): INTEGER; (* *returns the number of parameters associated with the statement, can only be called after the statement was prepared.* *)

PROCEDURE NumResultCols (s: SQL.Statement): INTEGER; (* *returns the number of columns in the result set associated with the statement.* *)

PROCEDURE PrimaryKeys (c: SQL.Connection; tableQual, tableOwner,
table: ARRAY OF CHAR): SQL.Statement; (* *returns a prepared statement holding the column names that comprise the primary key for a table. This function does not support returning primary keys from multiple tables in a single call.* *)

PROCEDURE ProcedureColumns (c: SQL.Connection; procQual, procOwnerPat,
procPat, colPat: ARRAY OF CHAR): SQL.Statement; (* *returns a prepared statement holding the list of input and output parameters, as well as the columns that make up the result set for the specified procedures.* *)

PROCEDURE Procedures (c: SQL.Connection; procQual, procOwnerPat,
procPat: ARRAY OF CHAR): SQL.Statement; (* *returns a prepared statement holding the list of procedure names stored in a specific data source. Procedure is a generic term used to describe an executable object or a named entity that can be invoked using input and output parameters, and which can return result sets similar to the result sets returned by SQL SELECT expressions.* *)

PROCEDURE SetCursorName (s: SQL.Statement; cursorName: ARRAY OF CHAR);
(* *associates the specified cursor name with the statement. The only ODBC SQL statements that use a cursor name are positioned update and delete (for example "UPDATE myTable ... WHERE CURRENT OF cursorName"). If the application does not call SetCursorName to define a cursor name, on execution of a SELECT statement the driver generates a name that begins with the letters SQL_CUR and does not exceed* 18 *characters in length. All cursor names associated with a connection must be unique.* *)

PROCEDURE SetCursorPos (s: SQL.Statement; rowIdx, option, lock: INTEGER); (* *sets the cursor position in a rowset and allows an application to refresh, update, delete, or add data*

```
    to the rowset. *)
  PROCEDURE SetScrollOptions (s: SQL.Statement; concurrency: INTEGER;
    rowsInKeyset: LONGINT; rowsInRowset: INTEGER); (* sets options that control the
    behaviour of cursors associated with the statement in the areas of concurrency control,
    sensitivity to changes made by other transactions, and rowset size. In ODBC 2.0
    SetScrollOptions has been superceded by the SQL_CURSOR_TYPE, SQL_CONCURRENCY,
    SQL_KEYSET_SIZE, and SQL_ROWSET_SIZE statement options. *)
  PROCEDURE SpecialColumns (c: SQL.Connection; colType: INTEGER;
    tableQual, tableOwner, table: ARRAY OF CHAR; scope,
    nullable: INTEGER): SQL.Statement; (* returns a prepared statement holding information
    about columns within the specified table, namely the optimal set of columns that uniquely
    identifies a row in the table (when colType = SQL_BEST_ROWID) or the columns
    that are automatically updated when any value in the row is updated by a transaction (when
    colType = SQL_ROWVER). *)
  PROCEDURE Statistics (c: SQL.Connection; tableQual, tableOwner,
    table: ARRAY OF CHAR; indexType, accuracy: INTEGER): SQL.Statement; (* returns
    a prepared statement holding a list of statistics about a single table and the indexes associated
    with the table. *)
  PROCEDURE TablePrivileges (c: SQL.Connection; tableQual, tableOwnerPat,
    tablePat: ARRAY OF CHAR): SQL.Statement; (* returns a prepared statement holding a
    list of tables and the privileges associated with each table. *)
  PROCEDURE Tables (c: SQL.Connection; tableQual, tableOwnerPat,
    tablePat, tableTypes: ARRAY OF CHAR): SQL.Statement; (* returns a prepared
    statement holding a list of all table names in the specified data source. tableTypes can hold
    a list of table types to match (e.g. "TABLE, VIEW" or "'TABLE', 'VIEW'"). *)

END SQLExt.
```

Anhang B

Beispielprogramme

B.1 Addresses

Im folgenden wird ein Programm zur Verwaltung einer kleinen Adreßkartei gezeigt, einmal unter Verwendung des Moduls EmbeddedSQL und einmal unter Verwendung des Moduls SQL.

B.1.1 Addresses - EmbeddedSQL

```
MODULE Addresses;

IMPORT ESQL := EmbeddedSQL, Files, Out, In;

TYPE
  Address = RECORD
    firstName, lastName, street: ARRAY 32 OF CHAR;
    town: ARRAY 20 OF CHAR;
    zipCode: INTEGER
  END ;

VAR
  conn: ESQL.Connection;
  connStr: ARRAY 256 OF CHAR;
  stat: ESQL.Statement;

PROCEDURE Open*;
BEGIN
  conn := ESQL.OpenUI(connStr, connStr)
END Open;

PROCEDURE Execute*;
  VAR sqlString: ARRAY 256 OF CHAR;
BEGIN
  In.Open; In.String(sqlString);
  IF In.Done THEN
```

```
    stat := conn.Prepare(sqlString);
    stat.Execute()
  END
END Execute;

PROCEDURE ListAll*;
  VAR adr: Address;
BEGIN
  stat := conn.Prepare("SELECT * FROM Addresses INTO :adr");
  stat.Execute();
  WHILE stat.Fetch() DO
    Out.String(adr.firstName); Out.Char(" "); Out.String(adr.lastName);
    Out.String(", ");
    IF ~stat.IsNull("adr.street") THEN Out.String(adr.street) END ;
    Out.String(", ");
    Out.Int(adr.zipCode, 0); Out.Char(" "); Out.String(adr.town); Out.Ln
  END
END ListAll;

PROCEDURE ListByName*; (** ^ | "lastName" *)
  VAR adr: Address; name: ARRAY 32 OF CHAR;
BEGIN
  In.Open; In.String(name);
  IF In.Done THEN
    stat := conn.Prepare("SELECT * FROM Addresses WHERE lastName = :name INTO :adr");
    stat.Execute();
    WHILE stat.Fetch() DO
      Out.String(adr.firstName); Out.Char(" "); Out.String(adr.lastName);
      Out.String(", ");
      IF ~stat.IsNull("adr.street") THEN Out.String(adr.street) END ;
      Out.String(", ");
      Out.Int(adr.zipCode, 0); Out.Char(" "); Out.String(adr.town); Out.Ln
    END
  END
END ListByName;

PROCEDURE Insert*; (** ^ | "firstName" "lastName" "street" "town" zipCode *)
  VAR adr: Address;
BEGIN
  In.Open;
  stat := conn.Prepare("INSERT INTO Addresses values (:adr)");
  REPEAT
    In.String(adr.firstName); In.String(adr.lastName); In.String(adr.street);
    In.String(adr.town); In.Int(adr.zipCode);
    IF adr.street = "NULL" THEN stat.SetNull("adr.street") END ;
    IF In.Done THEN stat.Execute() END
  UNTIL ~In.Done
END Insert;

BEGIN
  connStr := ""
END Addresses.
```

B.1.2 Addresses - SQL

```
MODULE Addresses;

IMPORT SQL, Files, Out, In;

VAR
  conn: SQL.Connection;
  connStr: ARRAY 256 OF CHAR;

PROCEDURE Open*;
  VAR s: ARRAY 256 OF CHAR;
BEGIN
  conn := SQL.OpenUI(connStr, s)
END Open;

PROCEDURE CreateTable*;
  VAR stat: SQL.Statement;
BEGIN
  stat := SQL.PrepareStatement(conn, "CREATE TABLE Friends (firstName TEXT(32),
  lastName TEXT(32), street TEXT(32), town TEXT(32), zipCode LONG)");
  SQL.Execute(stat);
  SQL.Commit(conn)
END CreateTable;

PROCEDURE InsertMyAddress*;
  VAR stat: SQL.Statement;
BEGIN
  stat := SQL.PrepareStatement(conn, "INSERT INTO Friends VALUES ('Christoph', 'Steindl',
  'Freistaedterstrasse 315', 'Linz', 4040)");
  SQL.Execute(stat);
  SQL.Commit(conn)
END InsertMyAddress;

PROCEDURE PutField* (VAR field: SQL.Field);
  VAR r: Files.Rider; ch: CHAR; f: SQL.Field;
BEGIN
  IF field.len = SQL.NullData THEN Out.String("NULL")
  ELSE
    f := field;
    WITH f: SQL.IntField DO Out.Int(f.i, 0)
    | f: SQL.StringField DO Out.String(f.str)
    | f: SQL.RealField DO
      CASE field.type.sqlType OF
        SQL.SQLFloat, SQL.SQLDouble: Out.LongReal(f.r, 0)
      | SQL.SQLReal: Out.Real(SHORT(f.r), 16)
      END
    | f: SQL.DateField DO Out.Int(f.year, 4); Out.Char("-"); Out.Int(f.month, 2);
      Out.Char("-"); Out.Int(f.day, 2)
    | f: SQL.TimeField DO Out.Int(f.hour, 2); Out.Char(":"); Out.Int(f.minute, 2);
      Out.Char(":"); Out.Int(f.second, 2)
    | f: SQL.TimeStampField DO Out.Int(f.year, 4); Out.Char("-"); Out.Int(f.month, 2);
```

```
      Out.Char("-"); Out.Int(f.day, 2); Out.Char(" "); Out.Int(f.hour, 2);
      Out.Char(":"); Out.Int(f.minute, 2); Out.Char(":"); Out.Int(f.second, 2);
      Out.Char(".");
      Out.Int(f.fraction, 0)
    | f: SQL.BooleanField DO
      IF f.b THEN Out.String("TRUE") ELSE Out.String("FALSE") END
    | f: SQL.FileField DO
      Out.String("Length of data: "); Out.Int(Files.Length(f.f), 0);
      IF field.type.sqlType = SQL.SQLLVarChar THEN
        Files.Set(r, f.f, 0); Files.Read(r, ch);
        WHILE ~r.eof DO Out.Char(ch); Files.Read(r, ch) END
      END
    ELSE (* SQL.BinaryField *)
    END
  END
END PutField;

PROCEDURE ListAll*;
  VAR stat: SQL.Statement; field: SQL.Field;
BEGIN
  stat := SQL.PrepareStatement(conn, "SELECT * FROM Friends");
  SQL.Execute(stat);
  IF (stat.ret = SQL.Success) & (stat.results # NIL) THEN
    Out.Ln; SQL.FirstField(stat.results, field);
    WHILE field # NIL DO
     Out.String(field.name); Out.Char(" ");
     SQL.NextField(field)
    END ;
    REPEAT
      SQL.Fetch(stat);
      IF stat.ret = SQL.Success THEN
        Out.Ln;
        SQL.FirstField(stat.results, field);
        WHILE field # NIL DO
          PutField(field); Out.Char(" ");
          SQL.NextField(field)
        END
      END
    UNTIL stat.ret # SQL.Success
  END ;
END ListAll;

PROCEDURE ListAddress*; (** ^ | "lastName" *)
  VAR name: ARRAY 32 OF CHAR; stat: SQL.Statement; field: SQL.Field;
    types: ARRAY 1 OF SQL.ParamDesc;
BEGIN
  In.Open; In.String(name);
  stat := SQL.PrepareStatement(conn, "SELECT * FROM Friends WHERE lastName = ?");
  types[0].type.sqlType := SQL.OberonChar;
  types[0].mapType.sqlType := SQL.OberonChar;
  types[0].inOut := SQL.InParam; types[0].name := "name";
  types[0].mapType.prec := 31;
```

```
  SQL.BindParameters(stat, types, 1);
  SQL.FindField(stat.params, "name", field);
  COPY(name, field(SQL.StringField).str);
  SQL.Execute(stat);
  IF (stat.ret = SQL.Success) & (stat.results # NIL) THEN
    Out.Ln; SQL.FirstField(stat.results, field);
    WHILE field # NIL DO
     Out.String(field.name); Out.Char(" ");
     SQL.NextField(field)
    END ;
    REPEAT
      SQL.Fetch(stat);
      IF stat.ret = SQL.Success THEN
        Out.Ln;
        SQL.FirstField(stat.results, field);
        WHILE field # NIL DO
          PutField(field); Out.Char(" ");
          SQL.NextField(field)
        END
      END
    UNTIL stat.ret # SQL.Success
  END
END ListAddress;

PROCEDURE Insert*; (** ^ | "firstName" "lastName" "street" "town" zipCode *)
  VAR firstName, lastName, street, town: ARRAY 32 OF CHAR; zipCode: LONGINT;
    stat: SQL.Statement; field: SQL.Field; types: ARRAY 5 OF SQL.ParamDesc;
    type, mapType: SQL.Type;
BEGIN
  In.Open; In.String(firstName); In.String(lastName); In.String(street);
  In.String(town); In.LongInt(zipCode);
  stat := SQL.PrepareStatement(conn, "INSERT INTO Friends VALUES (?, ?, ?, ?, ?)");
  type.sqlType := SQL.OberonChar; type.prec := 32; type.scale := 0;
  IF ~SQL.Map(conn, type, mapType) THEN HALT(69) END ;
  types[0].type := type; types[0].mapType := mapType; types[0].inOut := SQL.InParam;
  types[1] := types[0]; types[2] := types[0]; types[3] := types[0];
  types[0].name := "firstName"; types[1].name := "lastName";
  types[2].name := "street"; types[3].name := "town";
  type.sqlType := SQL.OberonLongInt; type.prec := 0; type.scale := 0;
  IF ~SQL.Map(conn, type, mapType) THEN HALT(69) END ;
  types[4].type := type; types[4].mapType := mapType; types[4].inOut := SQL.InParam;
  types[4].name := "zipCode";
  SQL.BindParameters(stat, types, 5);
  SQL.FindField(stat.params, "firstName", field);
  COPY(firstName, field(SQL.StringField).str);
  SQL.FindField(stat.params, "lastName", field);
  COPY(lastName, field(SQL.StringField).str);
  SQL.FindField(stat.params, "street", field);
  COPY(street, field(SQL.StringField).str);
  SQL.FindField(stat.params, "town", field);
  COPY(town, field(SQL.StringField).str);
  SQL.FindField(stat.params, "zipCode", field);
```

```
  field(SQL.IntField).i := zipCode;
  SQL.Execute(stat);
  SQL.Commit(conn)
END Insert;

BEGIN
  connStr := ""
END Addresses.
```

B.2 ODBCBench

ODBCBench ist ein portables Beispielprogramm. Es funktioniert ohne Änderungen auf den Oberon-Systemen V4 und V3. Um das Programm auch unter Oberon/F übersetzen zu können sind lediglich Änderungen aufgrund der anderen Schnittstelle des Moduls Files notwendig.

Die wesentlichen Schritte, um Portabilität zu erreichen sind das Ermitteln der von der jeweiligen Datenbank unterstützten Datentypen und das Umsetzen der gewünschten auf die unterstützten Datentypen. Weiters wird in diesem Programm gezeigt, wie man lange (dateibasierte) Daten in eine Tabelle einfügen und wieder auslesen kann.

```
MODULE ODBCBench;

IMPORT SQL, SQLExt, In, Out, Files, Strings;

CONST
  nofRows = 100;

VAR
  conn: SQL.Connection;
  connStr: ARRAY 255 OF CHAR;
  mapType, mapScale: ARRAY 3 OF INTEGER; mapPrec: ARRAY 3 OF LONGINT;
  f: Files.File;

PROCEDURE LongIntValueOf (f: SQL.Field): LONGINT;
  VAR i, tmp: LONGINT;
BEGIN
  WITH f: SQL.IntField DO RETURN f.i
  | f: SQL.StringField DO i := 0; tmp := 0;
      REPEAT
        tmp := tmp * 10 + ORD(f.str[i]) - ORD("0"); INC(i)
      UNTIL f.str[i] = 0X;
      RETURN tmp
  | f: SQL.RealField DO RETURN ENTIER(f.r)
  END
END LongIntValueOf;

PROCEDURE Disconnect*;
```

```
BEGIN conn := NIL END Disconnect;

PROCEDURE Connect*;
BEGIN
  IF conn # NIL THEN Disconnect END ;
  conn := SQL.OpenUI(connStr, connStr);
  IF ~SQL.Map(conn, SQL.OberonLongInt, 0, 0, mapType[0], mapPrec[0], mapScale[0]) OR
    ~SQL.Map(conn, SQL.OberonChar, 32, 0, mapType[1], mapPrec[1], mapScale[1]) OR
    ~SQL.Map(conn, SQL.SQLLVarChar, 50000, 0, mapType[2], mapPrec[2], mapScale[2]) THEN
    HALT(44)
  END
END Connect;

PROCEDURE DropTables*;
  VAR stat: SQL.Statement;
BEGIN
  stat := SQL.PrepareStatement(conn, "DROP table Bench1");
  SQL.Execute(stat);
  stat := SQL.PrepareStatement(conn, "DROP table Bench2");
  SQL.Execute(stat);
  stat := SQL.PrepareStatement(conn, "DROP table Bench3");
  SQL.Execute(stat);
END DropTables;

PROCEDURE CreateTables*;
  VAR stat: SQL.Statement; str: ARRAY 256 OF CHAR;
BEGIN
  str := "CREATE TABLE Bench1 (i ";
  SQLExt.AppendType(conn, str, mapType[0], mapPrec[0], mapScale[0]);
  Strings.Append(", firstName ", str);
  SQLExt.AppendType(conn, str, mapType[1], mapPrec[1], mapScale[1]);
  Strings.Append(", val ", str);
  SQLExt.AppendType(conn, str, mapType[0], mapPrec[0], mapScale[0]);
  Strings.Append(")", str);
  stat := SQL.PrepareStatement(conn, str);
  SQL.Execute(stat);
  Out.Ln; Out.String(str);
  IF stat.ret # SQL.Success THEN Out.String(" not") END ;
  Out.String(" created.");
  str := "CREATE TABLE Bench2 (i ";
  SQLExt.AppendType(conn, str, mapType[0], mapPrec[0], mapScale[0]);
  Strings.Append(", lastName ", str);
  SQLExt.AppendType(conn, str, mapType[1], mapPrec[1], mapScale[1]);
  Strings.Append(")", str);
  stat := SQL.PrepareStatement(conn, str);
  SQL.Execute(stat);
  Out.Ln; Out.String(str);
  IF stat.ret # SQL.Success THEN Out.String(" not") END ;
  Out.String(" created.");
  str := "CREATE TABLE Bench3 (i ";
  SQLExt.AppendType(conn, str, mapType[0], mapPrec[0], mapScale[0]);
  Strings.Append(", longData ", str);
```

```
  SQLExt.AppendType(conn, str, mapType[2], mapPrec[2], mapScale[2]);
  Strings.Append(")", str);
  stat := SQL.PrepareStatement(conn, str);
  SQL.Execute(stat);
  Out.Ln; Out.String(str);
  IF stat.ret # SQL.Success THEN Out.String(" not") END ;
  Out.String(" created.")
END CreateTables;

PROCEDURE GenerateName(i: INTEGER; VAR s: ARRAY OF CHAR);
BEGIN
  s[2] := CHR((i MOD 10) + ORD("0"));
  s[1] := CHR((i DIV 10 MOD 10) + ORD("0"));
  s[0] := CHR((i DIV 100 MOD 10) + ORD("0"));
  s[3] := 0X
END GenerateName;

PROCEDURE InsertIntoBench1*;
  VAR types: ARRAY 3 OF SQL.ParamDesc; stat: SQL.Statement;
    i: INTEGER; idx, firstName, value: SQL.Field;
BEGIN
  types[0].oberonType := SQL.OberonLongInt; types[0].sqlType := mapType[0];
  types[0].precision := mapPrec[0]; types[0].scale := mapScale[0];
  types[0].inOut := SQL.InParam; types[0].name := "idx";
  types[1].oberonType := SQL.OberonChar; types[1].sqlType := mapType[1];
  types[1].precision := mapPrec[1]; types[1].scale := mapScale[1];
  types[1].inOut := SQL.InParam; types[1].name := "firstName";
  types[2] := types[0]; types[2].name := "value";
  stat := SQL.PrepareStatement(conn,
    "INSERT INTO Bench1 (i, firstName, val) values (?, ?, ?)");
  SQL.BindParameters(stat, types, 3);
  SQL.FindField(stat.params, "idx", idx);
  SQL.FindField(stat.params, "firstName", firstName);
  SQL.FindField(stat.params, "value", value);
  FOR i := 0 TO nofRows DO
    idx(SQL.IntField).i := i;
    GenerateName(i, firstName(SQL.StringField).str);
    value(SQL.IntField).i := LONG(i) * i;
    SQL.Execute(stat);
    IF stat.ret # SQL.Success THEN RETURN END
  END
END InsertIntoBench1;

PROCEDURE AppendToBench1*;
  VAR types: ARRAY 3 OF SQL.ParamDesc; stat: SQL.Statement;
    i: INTEGER; idx, firstName, value: SQL.Field;
BEGIN
  stat := SQL.PrepareStatement(conn, "SELECT MAX(i) FROM Bench1");
  SQL.Execute(stat);
  SQL.FirstField(stat.results, idx);
  SQL.Fetch(stat);
  i := SHORT(LongIntValueOf(idx)); INC(i);
```

```
  types[0].oberonType := SQL.OberonLongInt; types[0].sqlType := mapType[0];
  types[0].precision := mapPrec[0]; types[0].scale := mapScale[0];
  types[0].inOut := SQL.InParam; types[0].name := "idx";
  types[1].oberonType := SQL.OberonChar; types[1].sqlType := mapType[1];
  types[1].precision := mapPrec[1]; types[1].scale := mapScale[1];
  types[1].inOut := SQL.InParam; types[1].name := "firstName";
  types[2] := types[0]; types[2].name := "value";
  stat := SQL.PrepareStatement(conn,
    "INSERT INTO Bench1 (i, firstName, val) values (?, ?, ?)");
  SQL.BindParameters(stat, types, 3);
  SQL.FindField(stat.params, "idx", idx);
  SQL.FindField(stat.params, "firstName", firstName);
  SQL.FindField(stat.params, "value", value);
  REPEAT
    idx(SQL.IntField).i := i;
    GenerateName(i, firstName(SQL.StringField).str);
    value(SQL.IntField).i := LONG(i) * i;
    SQL.Execute(stat);
    IF stat.ret # SQL.Success THEN RETURN END ;
    INC(i)
  UNTIL i MOD nofRows = 0
END AppendToBench1;

PROCEDURE InsertIntoBench2*;
  VAR types: ARRAY 2 OF SQL.ParamDesc; stat: SQL.Statement;
    i: INTEGER; idx, lastName: SQL.Field;
BEGIN
  types[0].oberonType := SQL.OberonLongInt; types[0].sqlType := mapType[0];
  types[0].precision := mapPrec[0]; types[0].scale := mapScale[0];
  types[0].inOut := SQL.InParam; types[0].name := "idx";
  types[1].oberonType := SQL.OberonChar; types[1].sqlType := mapType[1];
  types[1].precision := mapPrec[1]; types[1].scale := mapScale[1];
  types[1].inOut := SQL.InParam; types[1].name := "lastName";
  stat := SQL.PrepareStatement(conn,
    "INSERT INTO Bench2 (i, lastName) values (?, ?)");
  SQL.BindParameters(stat, types, 2);
  SQL.FindField(stat.params, "idx", idx);
  SQL.FindField(stat.params, "lastName", lastName);
  FOR i := 0 TO nofRows DO
    idx(SQL.IntField).i := i;
    GenerateName(i, lastName(SQL.StringField).str);
    SQL.Execute(stat);
    IF stat.ret # SQL.Success THEN RETURN END
  END
END InsertIntoBench2;

PROCEDURE PutField* (VAR field: SQL.Field);
  VAR r: Files.Rider; ch: CHAR; f: SQL.Field;
BEGIN
  IF field.len = SQL.NullData THEN Out.String("NULL")
  ELSE
    f := field;
```

```
    WITH f: SQL.IntField DO Out.Int(f.i, 0)
    | f: SQL.StringField DO Out.String(f.str)
    | f: SQL.RealField DO
      CASE field.sqlType OF
        SQL.SQLFloat, SQL.SQLDouble: Out.LongReal(f.r, 16)
      | SQL.SQLReal: Out.Real(SHORT(f.r), 16)
      END
    | f: SQL.DateField DO Out.Int(f.year, 0); Out.Char(\-");
      Out.Int(f.month, 0); Out.Char(\-"); Out.Int(f.day, 0)
    | f: SQL.TimeField DO Out.Int(f.hour, 0); Out.Char(":");
      Out.Int(f.minute, 0); Out.Char(":"); Out.Int(f.second, 0)
    | f: SQL.TimeStampField DO Out.Int(f.year, 0); Out.Char(\-");
      Out.Int(f.month, 0); Out.Char(\-"); Out.Int(f.day, 0);
      Out.Char(" "); Out.Int(f.hour, 0); Out.Char(":");
      Out.Int(f.minute, 0); Out.Char(":"); Out.Int(f.second, 0);
      Out.Char("."); Out.Int(f.fraction, 0)
    | f: SQL.BooleanField DO
      IF f.b THEN Out.String("TRUE") ELSE Out.String("FALSE") END
    | f: SQL.FileField DO
      Out.String("Length of data: "); Out.Int(Files.Length(f.f), 0);
      IF field.sqlType = SQL.SQLLVarChar THEN
        Files.Set(r, f.f, 0); Files.Read(r, ch);
        WHILE ~r.eof DO Out.Char(ch); Files.Read(r, ch) END
      END
    ELSE (* SQL.BinaryField *)
    END
  END
END PutField;

PROCEDURE ShowResults* (s: SQL.Statement);
  VAR field: SQL.Field; rows: LONGINT;
BEGIN
  IF s.ret = SQL.Success THEN
    IF s.results # NIL THEN
      Out.Ln; SQL.FirstField(s.results, field);
      WHILE field # NIL DO
        Out.String(field.name); Out.Char(9X); SQL.NextField(field)
      END ;
      REPEAT
        SQL.Fetch(s);
        IF s.ret = SQL.Success THEN
          Out.Ln;
          SQL.FirstField(s.results, field);
          WHILE field # NIL DO
            PutField(field); Out.Char(9X);
            SQL.NextField(field)
          END
        END
      UNTIL s.ret # SQL.Success
    ELSE
      SQL.RowCount(s, rows);
      IF rows # -1 THEN
```

```
        Out.Ln; Out.String("Number of rows affected: "); Out.Int(rows, 0)
      END
    END
  ELSE
    SQL.ShowError(s.ret, "ShowResults", SQL.env, conn.dbc, s.stmt)
  END ;
END ShowResults;

PROCEDURE Execute*;
  VAR stat: SQL.Statement; str: ARRAY 256 OF CHAR;
BEGIN
  In.Open; In.String(str);
  stat := SQL.PrepareStatement(conn, str);
  SQL.Execute(stat);
  ShowResults(stat)
END Execute;

PROCEDURE ExecuteIntParam*;
  VAR stat: SQL.Statement; str: ARRAY 256 OF CHAR;
    types: ARRAY 1 OF SQL.ParamDesc; par1: SQL.Field;
BEGIN
  In.Open; In.String(str);
  types[0].oberonType := SQL.OberonLongInt; types[0].sqlType := mapType[0];
  types[0].precision := mapPrec[0]; types[0].scale := mapScale[0];
  types[0].inOut := SQL.InParam; types[0].name := "par1";
  stat := SQL.PrepareStatement(conn, str);
  SQL.BindParameters(stat, types, 1);
  SQL.FindField(stat.params, "par1", par1);
  In.LongInt(par1(SQL.IntField).i);
  SQL.Execute(stat);
  ShowResults(stat)
END ExecuteIntParam;

PROCEDURE ExecuteStringParam*;
  VAR stat: SQL.Statement; str: ARRAY 256 OF CHAR;
    types: ARRAY 1 OF SQL.ParamDesc; par1: SQL.Field;
BEGIN
  In.Open; In.String(str);
  types[0].oberonType := SQL.SQLVarChar; types[0].sqlType := mapType[1];
  types[0].precision := mapPrec[1]; types[0].scale := mapScale[1];
  types[0].inOut := SQL.InParam; types[0].name := "par1";
  stat := SQL.PrepareStatement(conn, str);
  SQL.BindParameters(stat, types, 1);
  SQL.FindField(stat.params, "par1", par1);
  In.String(par1(SQL.StringField).str);
  SQL.Execute(stat);
  ShowResults(stat)
END ExecuteStringParam;

PROCEDURE CreateLongData*;
  VAR r: Files.Rider; i: INTEGER;
BEGIN
```

```
  f := Files.New(""); Files.Set(r, f, 0);
  FOR i := 0 TO 19999 DO Files.Write(r, CHR((i MOD 32) + ORD(" "))) END ;
END CreateLongData;

PROCEDURE VerifyLongData(f: Files.File);
  VAR r: Files.Rider; ch: CHAR; i: INTEGER;
BEGIN
  Files.Set(r, f, 0);
  FOR i := 0 TO 19999 DO
    Files.Read(r, ch);
    IF ch # CHR((i MOD 32) + ORD(" ")) THEN
      Out.String("wrong character in file");
      RETURN
    END
  END
END VerifyLongData;

PROCEDURE CopyFile(fIn, fOut: Files.File);
  VAR r, w: Files.Rider; buffer: ARRAY 1024 OF CHAR; pos: LONGINT;
BEGIN
  Files.Set(r, fIn, 0); Files.Set(w, fOut, 0);
  REPEAT
    Files.ReadBytes(r, buffer, 1024);
    Files.WriteBytes(w, buffer, 1024 - r.res)
  UNTIL r.eof;
END CopyFile;

PROCEDURE InsertLongData*;
  VAR types: ARRAY 2 OF SQL.ParamDesc; stat: SQL.Statement;
    i: INTEGER; idx, longData: SQL.Field;
BEGIN
  types[0].oberonType := SQL.OberonLongInt; types[0].sqlType := mapType[0];
  types[0].precision := mapPrec[0]; types[0].scale := mapScale[0];
  types[0].inOut := SQL.InParam; types[0].name := "idx";
  types[1].oberonType := SQL.SQLLVarChar; types[1].sqlType := mapType[2];
  types[1].inOut := SQL.InParam; types[1].name := "longData";
  types[1].precision := mapPrec[2]; types[1].scale := mapScale[2];
  stat := SQL.PrepareStatement(conn,
    "INSERT INTO Bench3 (i, longData) values (?, ?)");
  SQL.BindParameters(stat, types, 2);
  SQL.FindField(stat.params, "idx", idx);
  SQL.FindField(stat.params, "longData", longData);
  FOR i := 0 TO 10 DO
    Out.Ln; Out.String("Inserting row number "); Out.Int(i, 0);
    idx(SQL.IntField).i := i;
    CopyFile(f, longData(SQL.FileField).f);
    SQL.Execute(stat);
    IF stat.ret # SQL.Success THEN RETURN END
  END
END InsertLongData;

PROCEDURE GetLongData*;
```

```
  VAR stat: SQL.Statement;
    i: INTEGER; idx, longData: SQL.Field;
BEGIN
  stat := SQL.PrepareStatement(conn, "SELECT * FROM Bench3");
  SQL.FindField(stat.results, "i", idx);
  SQL.FindField(stat.results, "longData", longData);
  SQL.Execute(stat);
  IF stat.ret = SQL.Success THEN
    i := 0;
    REPEAT
      SQL.Fetch(stat);
      IF stat.ret = SQL.Success THEN
        Out.Ln; Out.String("Fetching row number "); Out.Int(i, 0);
        IF idx(SQL.IntField).i # i THEN
          Out.String("wrong index: "); Out.Int(idx(SQL.IntField).i, 0)
        END ;
        VerifyLongData(longData(SQL.FileField).f);
        INC(i)
      END
    UNTIL stat.ret # SQL.Success;
  END
END GetLongData;

BEGIN
  connStr := ""
END ODBCBench.
```

Anhang C

Konstantendeklarationen

Die Konstanten sind in den Dateien SQL.H und SQLEXT.H des ODBC Software Development Kit deklariert. Eine genaue Beschreibung der Bedeutung der einzelnen Optionen findet man in [ODBC94].

C.1 GetStatementOption und SetStatementOption

Für alle angeführten Optionen muß der Parameter info ein SQL.IntField sein.

ODBC-Version < 200
SQL_QUERY_TIMEOUT 0
  SQL_QUERY_TIMEOUT_DEFAULT 0
SQL_MAX_ROWS 1
  SQL_MAX_ROWS_DEFAULT 0
SQL_NOSCAN 2
  SQL_NOSCAN_OFF 0 unter ODBC 1.0: FALSE
  SQL_NOSCAN_ON 1 unter ODBC 1.0: TRUE
  SQL_NOSCAN_DEFAULT SQL_NOSCAN_OFF
SQL_MAX_LENGTH 3
  SQL_MAX_LENGTH_DEFAULT 0
SQL_ASYNC_ENABLE 4
  SQL_ASYNC_ENABLE_OFF 0
  SQL_ASYNC_ENABLE_ON 1
  SQL_ASYNC_ENABLE_DEFAULT SQL_ASYNC_ENABLE_OFF
SQL_BIND_TYPE 5
  SQL_BIND_BY_COLUMN 0

ODBC-Version ≥ 200
SQL_CURSOR_TYPE 6
  SQL_CURSOR_FORWARD_ONLY 0
  SQL_CURSOR_KEYSET_DRIVEN 1
  SQL_CURSOR_DYNAMIC 2
  SQL_CURSOR_STATIC 3
SQL_CONCURRENCY 7
  SQL_CONCUR_READ_ONLY 1

SQL_CONCUR_LOCK 2
SQL_CONCUR_ROWVER 3
SQL_CONCUR_VALUES 4
SQL_KEYSET_SIZE 8
SQL_KEYSET_SIZE_DEFAULT 0
SQL_ROWSET_SIZE 9
SQL_ROWSET_SIZE_DEFAULT 1
SQL_SIMULATE_CURSOR 10
SQL_SC_NON_UNIQUE 0
SQL_SC_TRY_UNIQUE 1
SQL_SC_UNIQUE 2
SQL_RETRIEVE_DATA 11
SQL_RD_OFF 0
SQL_RD_ON 1
SQL_RD_DEFAULT SQL_RD_ON
SQL_USE_BOOKMARKS 12
SQL_UB_OFF 0
SQL_UB_ON 1
SQL_UB_DEFAULT SQL_UB_OFF
SQL_GET_BOOKMARK 13 nur fr GetStmtOption
SQL_ROW_NUMBER 14 nur fr GetStmtOption

C.2 GetConnOption and SetConnOption

ODBC-Version < 200
SQL_ACCESS_MODE 101 IntField
SQL_MODE_READ_WRITE 0
SQL_MODE_READ_ONLY 1
SQL_MODE_DEFAULT SQL_MODE_READ_WRITE
SQL_AUTOCOMMIT 102 IntField
SQL_AUTOCOMMIT_OFF 0
SQL_AUTOCOMMIT_ON 1
SQL_AUTOCOMMIT_DEFAULT SQL_AUTOCOMMIT_ON
SQL_LOGIN_TIMEOUT 103 IntField
SQL_LOGIN_TIMEOUT_DEFAULT 15
SQL_OPT_TRACE 104 IntField
SQL_OPT_TRACE_OFF 0
SQL_OPT_TRACE_ON 1
SQL_OPT_TRACE_DEFAULT SQL_OPT_TRACE_OFF
SQL_OPT_TRACEFILE 105 StringField
SQL_OPT_TRACE_FILE_DEFAULT „\SQL.LOG“
SQL_TRANSLATE_DLL 106 StringField
SQL_TRANSLATE_OPTION 107 IntField
SQL_TXN_ISOLATION 108 IntField
SQL_TXN_READ_UNCOMMITTED 1H
SQL_TXN_READ_COMMITTED 2H
SQL_TXN_REPEATABLE_READ 4H
SQL_TXN_SERIALIZABLE 8H

SQL_TXN_VERSIONING 10H
SQL_CURRENT_QUALIFIER 109 StringField

ODBC-Version ≥ 200
SQL_ODBC_CURSORS 110 IntField
SQL_CUR_USE_IF_NEEDED 0
SQL_CUR_USE_ODBC 1
SQL_CUR_USE_DRIVER 2
SQL_CUR_DEFAULT SQL_CUR_USE_DRIVER
SQL_QUIET_MODE 111 IntField
SQL_PACKET_SIZE 112 IntField

C.3 GetInfo

ODBC-Version < 200
SQL_ACTIVE_CONNECTIONS 0 IntField
SQL_ACTIVE_STATEMENTS 1 IntField
SQL_DATA_SOURCE_NAME 2 StringField
SQL_DRIVER_HDBC 3 IntField
SQL_DRIVER_HENV 4 IntField
SQL_DRIVER_HSTMT 5 IntField
SQL_DRIVER_NAME 6 StringField
SQL_DRIVER_VER 7 StringField
SQL_FETCH_DIRECTION 8 IntField
SQL_FD_FETCH_NEXT 1H
SQL_FD_FETCH_FIRST 2H
SQL_FD_FETCH_LAST 4H
SQL_FD_FETCH_PRIOR 8H
SQL_FD_FETCH_ABSOLUTE 10H
SQL_FD_FETCH_RELATIVE 20H
SQL_FD_FETCH_RESUME 40H
SQL_FD_FETCH_BOOKMARK 80H
SQL_ODBC_API_CONFORMANCE 9 IntField
SQL_OAC_NONE 0H
SQL_OAC_LEVEL1 1H
SQL_OAC_LEVEL2 2H
SQL_ODBC_VER 10 StringField
SQL_ROW_UPDATES 11 StringField
SQL_ODBC_SAG_CLI_CONFORMANCE 12 IntField
SQL_OSCC_NOT_COMPLIANT 0H
SQL_OSCC_COMPLIANT 1H
SQL_SERVER_NAME 13 StringField
SQL_SEARCH_PATTERN_ESCAPE 14 StringField
SQL_ODBC_SQL_CONFORMANCE 15 IntField
SQL_OSC_MINIMUM 0H
SQL_OSC_CORE 1H
SQL_OSC_EXTENDED 2H
SQL_DBMS_NAME 17 StringField

SQL_DBMS_VER 18 StringField
SQL_ACCESSIBLE_TABLES 19 StringField
"Y" if the user has SELECT privileges on all tables returned by Tables,
"N" if there may be tables that the user cannot access.
SQL_ACCESSIBLE_PROCEDURES 20 StringField
"Y" if the user can execute all procedures returned by Procedures,
"N" if there may be procedures that the user cannot execute.
SQL_PROCEDURES 21 StringField
SQL_CONCAT_NULL_BEHAVIOR 22 IntField
SQL_CB_NULL 0H
SQL_CB_NON_NULL 1H
SQL_CURSOR_COMMIT_BEHAVIOR 23 IntField
SQL_CB_DELETE 0H
SQL_CB_CLOSE 1H
SQL_CB_PRESERVE 2H
SQL_CURSOR_ROLLBACK_BEHAVIOR 24 IntField
SQL_CB_DELETE 0H
SQL_CB_CLOSE 1H
SQL_CB_PRESERVE 2H
SQL_DATA_SOURCE_READ_ONLY 25 StringField
SQL_DEFAULT_TXN_ISOLATION 26 IntField
SQL_TXN_READ_UNCOMMITTED 1H
SQL_TXN_READ_COMMITTED 2H
SQL_TXN_REPEATABLE_READ 4H
SQL_TXN_SERIALIZABLE 8H
SQL_TXN_VERSIONING 10H
SQL_EXPRESSIONS_IN_ORDERBY 27 StringField
SQL_IDENTIFIER_CASE 28 IntField
SQL_IC_UPPER 1H
SQL_IC_LOWER 2H
SQL_IC_SENSITIVE 3H
SQL_IC_MIXED 4H
SQL_IDENTIFIER_QUOTE_CHAR 29 StringField
SQL_MAX_COLUMN_NAME_LEN 30 IntField
SQL_MAX_CURSOR_NAME_LEN 31 IntField
SQL_MAX_OWNER_NAME_LEN 32 IntField
SQL_MAX_PROCEDURE_NAME_LEN 33 IntField
SQL_MAX_QUALIFIER_NAME_LEN 34 IntField
SQL_MAX_TABLE_NAME_LEN 35 IntField
SQL_MULT_RESULT_SETS 36 StringField
SQL_MULTIPLE_ACTIVE_TXN 37 StringField
SQL_OUTER_JOINS 38 StringField
SQL_OWNER_TERM 39 StringField
SQL_PROCEDURE_TERM 40 StringField
SQL_QUALIFIER_NAME_SEPARATOR 41 StringField
SQL_QUALIFIER_TERM 42 StringField
SQL_SCROLL_CONCURRENCY 43 IntField
SQL_SCCO_READ_ONLY 1H
SQL_SCCO_LOCK 2H
SQL_SCCO_OPT_ROWVER 4H
SQL_SCCO_OPT_VALUES 8H
SQL_SCROLL_OPTIONS 44 IntField

SQL_SO_FORWARD_ONLY 1H
SQL_SO_KEYSET_DRIVEN 2H
SQL_SO_DYNAMIC 4H
SQL_SO_MIXED 8H
SQL_SO_STATIC 10H
SQL_TABLE_TERM 45 StringField
SQL_TXN_CAPABLE 46 IntField
SQL_TC_NONE 0H
SQL_TC_DML 1H
SQL_TC_ALL 2H
SQL_TC_DDL_COMMIT 3H
SQL_TC_DDL_IGNORE 4H
SQL_USER_NAME 47 StringField
SQL_CONVERT_FUNCTIONS 48 IntField
SQL_CVT_CHAR 1H
SQL_CVT_NUMERIC 2H
SQL_CVT_DECIMAL 4H
SQL_CVT_INTEGER 8H
SQL_CVT_SMALLINT 10H
SQL_CVT_FLOAT 20H
SQL_CVT_REAL 40H
SQL_CVT_DOUBLE 80H
SQL_CVT_VARCHAR 100H
SQL_CVT_LONGVARCHAR 200H
SQL_CVT_BINARY 400H
SQL_CVT_VARBINARY 800H
SQL_CVT_BIT 1000H
SQL_CVT_TINYINT 2000H
SQL_CVT_BIGINT 4000H
SQL_CVT_DATE 8000H
SQL_CVT_TIME 10000H
SQL_CVT_TIMESTAMP 20000H
SQL_CVT_LONGVARBINARY 40000H
SQL_NUMERIC_FUNCTIONS 49 IntField
SQL_FN_NUM_ABS 1H
SQL_FN_NUM_ACOS 2H
SQL_FN_NUM_ASIN 4H
SQL_FN_NUM_ATAN 8H
SQL_FN_NUM_ATAN2 10H
SQL_FN_NUM_CEILING 20H
SQL_FN_NUM_COS 40H
SQL_FN_NUM_COT 80H
SQL_FN_NUM_EXP 100H
SQL_FN_NUM_FLOOR 200H
SQL_FN_NUM_LOG 400H
SQL_FN_NUM_MOD 800H
SQL_FN_NUM_SIGN 1000H
SQL_FN_NUM_SIN 2000H
SQL_FN_NUM_SQRT 4000H
SQL_FN_NUM_TAN 8000H
SQL_FN_NUM_PI 10000H
SQL_FN_NUM_RAND 20000H

SQL_FN_NUM_DEGREES 40000H
SQL_FN_NUM_LOG10 80000H
SQL_FN_NUM_POWER 100000H
SQL_FN_NUM_RADIANS 200000H
SQL_FN_NUM_ROUND 400000H
SQL_FN_NUM_TRUNCATE 800000H
SQL_STRING_FUNCTIONS 50 IntField
SQL_FN_STR_CONCAT 1H
SQL_FN_STR_INSERT 2H
SQL_FN_STR_LEFT 4H
SQL_FN_STR_LTRIM 8H
SQL_FN_STR_LENGTH 10H
SQL_FN_STR_LOCATE 20H
SQL_FN_STR_LCASE 40H
SQL_FN_STR_REPEAT 80H
SQL_FN_STR_REPLACE 100H
SQL_FN_STR_RIGHT 200H
SQL_FN_STR_RTRIM 400H
SQL_FN_STR_SUBSTRING 800H
SQL_FN_STR_UCASE 1000H
SQL_FN_STR_ASCII 2000H
SQL_FN_STR_CHAR 4000H
SQL_FN_STR_DIFFERENCE 8000H
SQL_FN_STR_LOCATE_2 10000H
SQL_FN_STR_SOUNDEX 20000H
SQL_FN_STR_SPACE 40000H
SQL_SYSTEM_FUNCTIONS 51 IntField
SQL_FN_SYS_USERNAME 1H
SQL_FN_SYS_DBNAME 2H
SQL_FN_SYS_IFNULL 4H
SQL_TIMEDATE_FUNCTIONS 52 IntField
SQL_FN_TD_NOW 1H
SQL_FN_TD_CURDATE 2H
SQL_FN_TD_DAYOFMONTH 4H
SQL_FN_TD_DAYOFWEEK 8H
SQL_FN_TD_DAYOFYEAR 10H
SQL_FN_TD_MONTH 20H
SQL_FN_TD_QUARTER 40H
SQL_FN_TD_WEEK 80H
SQL_FN_TD_YEAR 100H
SQL_FN_TD_CURTIME 200H
SQL_FN_TD_HOUR 400H
SQL_FN_TD_MINUTE 800H
SQL_FN_TD_SECOND 1000H
SQL_FN_TD_TIMESTAMPADD 2000H
SQL_FN_TD_TIMESTAMPDIFF 4000H
SQL_FN_TD_DAYNAME 8000H
SQL_FN_TD_MONTHNAME 10000H
SQL_CONVERT_BIGINT 53 IntField
SQL_CONVERT_BINARY 54 IntField
SQL_CONVERT_BIT 55 IntField
SQL_CONVERT_CHAR 56 IntField

SQL_CONVERT_DATE 57 IntField
SQL_CONVERT_DECIMAL 58 IntField
SQL_CONVERT_DOUBLE 59 IntField
SQL_CONVERT_FLOAT 60 IntField
SQL_CONVERT_INTEGER 61 IntField
SQL_CONVERT_LONGVARCHAR 62 IntField
SQL_CONVERT_NUMERIC 63 IntField
SQL_CONVERT_REAL 64 IntField
SQL_CONVERT_SMALLINT 65 IntField
SQL_CONVERT_TIME 66 IntField
SQL_CONVERT_TIMESTAMP 67 IntField
SQL_CONVERT_TINYINT 68 IntField
SQL_CONVERT_VARBINARY 69 IntField
SQL_CONVERT_VARCHAR 70 IntField
SQL_CONVERT_LONGVARBINARY 71 IntField
SQL_TXN_ISOLATION_OPTION 72 IntField
  SQL_TXN_READ_UNCOMMITTED 1H
  SQL_TXN_READ_COMMITTED 2H
  SQL_TXN_REPEATABLE_READ 4H
  SQL_TXN_SERIALIZABLE 8H
  SQL_TXN_VERSIONING 10H
SQL_ODBC_SQL_OPT_IEF 73 StringField
SQL_CORRELATION_NAME 74 IntField
  SQL_CN_NONE 0H
  SQL_CN_DIFFERENT 1H
  SQL_CN_ANY 2H
SQL_NON_NULLABLE_COLUMNS 75 IntField
  SQL_NNC_NULL 0H
  SQL_NNC_NON_NULL 1H

ODBC-Version ≥ 200
SQL_DRIVER_HLIB 76 IntField
SQL_DRIVER_ODBC_VER 77 StringField
SQL_LOCK_TYPES 78 IntField
  SQL_LCK_NO_CHANGE 1H
  SQL_LCK_EXCLUSIVE 2H
  SQL_LCK_UNLOCK 4H
SQL_POS_OPERATIONS 79 IntField
  SQL_POS_POSITION 1H
  SQL_POS_REFRESH 2H
  SQL_POS_UPDATE 4H
  SQL_POS_DELETE 8H
  SQL_POS_ADD 10H
SQL_POSITIONED_STATEMENTS 80 IntField
  SQL_PS_POSITIONED_DELETE 1H
  SQL_PS_POSITIONED_UPDATE 2H
  SQL_PS_SELECT_FOR_UPDATE 4H
SQL_GETDATA_EXTENSIONS 81 IntField
  SQL_GD_ANY_COLUMN 1H
  SQL_GD_ANY_ORDER 2H
  SQL_GD_BLOCK 4H
  SQL_GD_BOUND 8H

SQL_BOOKMARK_PERSISTENCE 82 IntField
SQL_BP_CLOSE 1H
SQL_BP_DELETE 2H
SQL_BP_DROP 4H
SQL_BP_TRANSACTION 8H
SQL_BP_UPDATE 10H
SQL_BP_OTHER_HSTMT 20H
SQL_BP_SCROLL 40H
SQL_STATIC_SENSITIVITY 83 IntField
SQL_SS_ADDITIONS 1H
SQL_SS_DELETIONS 2H
SQL_SS_UPDATES 4H
SQL_FILE_USAGE 84 IntField
SQL_FILE_NOT_SUPPORTED 0H
SQL_FILE_TABLE 1H
SQL_FILE_QUALIFIER 2H
SQL_NULL_COLLATION 85 IntField
SQL_NC_HIGH 0H
SQL_NC_LOW 1H
SQL_NC_START 2H
SQL_NC_END 4H
SQL_ALTER_TABLE 86 IntField as bitmask of supported clauses
SQL_AT_ADD_COLUMN 1H
SQL_AT_DROP_COLUMN 2H
SQL_COLUMN_ALIAS 87 StringField
"Y" if the data source supports column aliases; otherwise "N".
SQL_GROUP_BY 88 IntField
SQL_GB_NOT_SUPPORTED 0H
SQL_GB_GROUP_BY_EQUALS_SELECT 1H
SQL_GB_GROUP_BY_CONTAINS_SELECT 2H
SQL_GB_NO_RELATION 3H
SQL_KEYWORDS 89 StringField
SQL_ORDER_BY_COLUMNS_IN_SELECT 90 StringField
SQL_OWNER_USAGE 91 IntField
SQL_OU_DML_STATEMENTS 1H
SQL_OU_PROCEDURE_INVOCATION 2H
SQL_OU_TABLE_DEFINITION 4H
SQL_OU_INDEX_DEFINITION 8H
SQL_OU_PRIVILEGE_DEFINITION 10H
SQL_QUALIFIER_USAGE 92 IntField
SQL_QU_DML_STATEMENTS 1H
SQL_QU_PROCEDURE_INVOCATION 2H
SQL_QU_TABLE_DEFINITION 4H
SQL_QU_INDEX_DEFINITION 8H
SQL_QU_PRIVILEGE_DEFINITION 10H
SQL_QUOTED_IDENTIFIER_CASE 93 IntField
SQL_IC_UPPER 1H
SQL_IC_LOWER 2H
SQL_IC_SENSITIVE 3H
SQL_IC_MIXED 4H
SQL_SPECIAL_CHARACTERS 94 StringField
SQL_SUBQUERIES 95 IntField

SQL_SQ_COMPARISON 1H
SQL_SQ_EXISTS 2H
SQL_SQ_IN 4H
SQL_SQ_QUANTIFIED 8H
SQL_SQ_CORRELATED_SUBQUERIES 10H
SQL_UNION 96 IntField
SQL_U_UNION 1H
SQL_U_UNION_ALL 2H
SQL_MAX_COLUMNS_IN_GROUP_BY 97 IntField
SQL_MAX_COLUMNS_IN_INDEX 98 IntField
SQL_MAX_COLUMNS_IN_ORDER_BY 99 IntField
SQL_MAX_COLUMNS_IN_SELECT 100 IntField
SQL_MAX_COLUMNS_IN_TABLE 101 IntField
SQL_MAX_INDEX_SIZE 102 IntField
SQL_MAX_ROW_SIZE_INCLUDES_LONG 103 StringField
SQL_MAX_ROW_SIZE 104 IntField
SQL_MAX_STATEMENT_LEN 105 IntField
SQL_MAX_TABLES_IN_SELECT 106 IntField
SQL_MAX_USER_NAME_LEN 107 IntField
SQL_MAX_CHAR_LITERAL_LEN 108 IntField
SQL_TIMEDATE_ADD_INTERVALS 109 IntField
SQL_FN_TSI_FRAC_SECOND 1H
SQL_FN_TSI_SECOND 2H
SQL_FN_TSI_MINUTE 4H
SQL_FN_TSI_HOUR 8H
SQL_FN_TSI_DAY 10H
SQL_FN_TSI_WEEK 20H
SQL_FN_TSI_MONTH 40H
SQL_FN_TSI_QUARTER 80H
SQL_FN_TSI_YEAR 100H
SQL_TIMEDATE_DIFF_INTERVALS 110 IntField
SQL_FN_TSI_FRAC_SECOND 1H
SQL_FN_TSI_SECOND 2H
SQL_FN_TSI_MINUTE 4H
SQL_FN_TSI_HOUR 8H
SQL_FN_TSI_DAY 10H
SQL_FN_TSI_WEEK 20H
SQL_FN_TSI_MONTH 40H
SQL_FN_TSI_QUARTER 80H
SQL_FN_TSI_YEAR 100H
SQL_NEED_LONG_DATA_LEN 111 StringField
SQL_MAX_BINARY_LITERAL_LEN 112 IntField
SQL_LIKE_ESCAPE_CLAUSE 113 StringField
SQL_QUALIFIER_LOCATION 114 IntField
SQL_QL_START 1HL
SQL_QL_END 2HL

ODBC-Version ≥ 201
SQL_OJ_CAPABILITIES 65002 temporärer Wert bis ODBC 3.0
SQL_OJ_LEFT 1H
SQL_OJ_RIGHT 2H
SQL_OJ_FULL 4H

SQL_OJ_NESTED 8H
SQL_OJ_NOT_ORDERED 10H
SQL_OJ_INNER 20H
SQL_OJ_ALL_COMPARISON_OPS 40H

C.4 SpecialColumns

Werte für colType:
SQL_BEST_ROWID 1
SQL_ROWVER 2

Werte für scope:
SQL_SCOPE_CURROW 0
SQL_SCOPE_TRANSACTION 1
SQL_SCOPE_SESSION 2

Werte für nullable:
SQL_NO_NULLS 0
SQL_NULLABLE 1
SQL_NULLABLE_UNKNOWN 2

Literaturverzeichnis

[Chen76] P. P. Chen. *Entity-Relationship Model: Towards a Unified View of Data*, ACM Transactions on Database Systems, 1 1976, 9-36

[Codd70] E. F. Codd. *A Relational Model of Data for Large Shared Data Banks*, CACM 13, June 1970, 377-387

[Dubbel] W. Beitz und K.-H. Küttner. *Dubbel, Taschenbuch für den Maschinenbau.* 17. Auflage, Springer-Verlag, 1990, ISBN 3-540-52381-2

[MöK96] Hanspeter Mössenböck, Kai Koskimies. *Active Text for Structuring and Understanding Source Code.* SOFTWARE - Practice and Experience, 26(7): 833-850, July 1996.

[Heu95] A. Heuer, G. Saake. *Datenbanken.* International Thomson Publishing GmbH, 1995, ISBN 3-929821-31-1

[Moe93] Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2.* Springer-Verlag, 1993, ISBN 3-540-56411-X

[Pou95] D. Pountain. *The Oberon/F System*, Byte, January 1995

[PLSQL] S. Feuerstein. *Oracle PL/SQL Programming*, O'Reilly & Associates, Inc., September 1995, ISBN 1-56592-142-9

[ODBC94] *Microsoft Open Database Connectivity Software Development Kit Version 2.0.* Microsoft Press, 1994, ISBN 1-55615-658-8

[StM96] Christoph Steindl, Hanspeter Mössenböck. *Reflection in Oberon.* Technical Report 8, 1996, Abteilung für Systemsoftware, Universität

[St96] Christoph Steindl. *Is Interoperability Achievable with ODBC?*, CON'96 Future Information Technologies, 11^{th} Austrian-Hungarian Informatics Conference Proceedings, November 1996, ISBN 3-486-24059-5.

[SQL2] Jim Melton, Alan R. Simon. *Understanding the new SQL: a complete guide.* Morgan Kaufmann Publishers, Inc., 1993, ISBN 1-55860-245-3

[Tem94] Josef Templ. *Metaprogramming in Oberon.* Diss. ETH No 10655, 1994

[Ull88] Jeffrey Ullman. *Principles of database and knowledgebase systems.* Volume 1, Computer Science Press, 1988, ISBN 0-7167-8158-1

[WiG89] N. Wirth, J. Gutknecht. *The Oberon System.* SOFTWARE - Practice and Experience, 19(9), September 1989.