

# Anwendungsprogrammierung in (objekt-)relationalen Datenbanksystemen

7.1 Die Datenbankprogrammiersprache Oracle-PL/SQL .....	352
7.2 Die Datenbankprogrammiersprache bei MySQL ..	386
7.3 Aktive Datenbanksysteme .....	402
Zusammenfassung .....	430
Weiterführende Literatur .....	431
Übungsaufgaben .....	432

» Dieses Kapitel befasst sich mit den Möglichkeiten, zu einem Datenmodell und einem Datenbankschema eine Benutzerschnittstelle zu gestalten, die im ANSI-3-Ebenen-Modell in der externen Ebene vorgesehen ist. Grundsätzlich geht es um die Koppelung von SQL mit einer prozeduralen Sprache. Wegen der mengenorientierten Verarbeitung in SQL und der Einzelverarbeitung von Datensätzen in einer prozeduralen Programmiersprache taucht das grundsätzliche Problem der Verbindung dieser beiden Ansätze auf. In der Literatur spricht man von „Impedance Mismatch“. Diesem Problem versucht man z.B. mit dem Cursor-Ansatz oder mit ResultSets bei JDBC zu begegnen. Bei diesem Ansatz werden die mengenorientierten Ergebnisse mittels Schleifen durchlaufen.

Grundsätzlich bestehen bei der Kopplung von SQL mit einer prozeduralen Sprache drei Möglichkeiten:

### 1 Erweiterung der Datenbanksprache um Programmierkonstrukte:

Bei diesem Ansatz spricht man auch von einer sogenannten „4th Generation Language“. Die Sprache SQL wird um prozedurale Bestandteile erweitert. Der SQL-Standard selbst hat erstmalig 1996 in einer SQL2-Erweiterung Vorgaben für seine sogenannten SQL/PSM-Prozeduren (persistent stored modules) gemacht, die dann 1999 durch die SQL-invoked routines ersetzt wurden. Herstellerspezifische Beispiele für eine solche prozedurale Erweiterung sind PL/SQL (procedural language) von Oracle oder die gespeicherten Routinen bei MySQL5. Alle Hersteller gängiger großer relationaler Datenbanksysteme haben ähnliche Sprachen, die mehr oder weniger vollständig dem Standard entsprechen. Ein Grund für Abweichungen ist oft ein historischer: Die SQL-invoked routines wurden sehr viel später in den Standard aufgenommen, nachdem z.B. die Sprache PL/SQL und andere schon jahrelang in der Praxis angewendet wurden. Anders sieht es bei MySQL aus, wo dieses Konzept erst ganz neu mit der Version 5 eingeführt wurde. Dort hat man die späte Implementierung als Chance genutzt, sich an den SQL2003-Standard zu halten. Wir werden im Abschnitt 7.1 den Oracle-Ansatz vorstellen und den MySQL-Ansatz im Abschnitt 7.2.

### 2 Einbettung der Datenbanksprache in eine Programmiersprache (Embedded SQL, ESQL):

Lange schon gibt es Embedded C, Embedded Cobol, Embedded Pascal etc. Wegen der raschen Verbreitung von Java ist auch Embedded Java dazugekommen. Man spricht hier von SQLJ, einem Standard, der in die Normierungsbestrebungen unter SQL 1999 aufgenommen wurde. Grundsätzlich gibt es hier zwei Vorgehensweisen:

- **CLI oder Call Level Interface:** Hierzu gehören die Ansätze ODBC, JDBC, PHP und Perl. Es werden Funktionsbibliotheken aufgerufen, die über CLI standardisiert sind.
- **Einbettung der SQL-Datenbanksprache in Java:** Bei diesem Ansatz werden die SQL-Anweisungen schon beim Übersetzen geparst und in Java-Anweisungen übersetzt. Problem ist hier die Abbildung von Tupelmengen auf die Datentypen der Programmiersprache. Die einzelnen Attribute eines Datensatzes werden auf die Datentypen der Programmiersprache abgebildet. Als typische Einbettung haben wir in Kapitel 6 SQLJ behandelt.

### 3 Erweiterung einer Programmiersprache um Datenbankkonstrukte:

In diesem Zusammenhang sind persistente Programmiersprachen, z.B. PASCAL/R, zu nennen. Diesen Typ der Koppelung von SQL und einer prozeduralen Sprache werden wir nicht behandeln.

Neben diesen passiven Programmen, die für ihre Ausführung explizit vom Anwender aufgerufen werden müssen, gibt es auch noch die Möglichkeit, aktive Programme in Form sogenannter aktiver Regeln (Trigger) in der Datenbank anzulegen. Kennzeichnend bei diesen „Programmen“ ist, dass sie automatisch ausgeführt werden, wenn ein zuvor spezifiziertes Ereignis eingetreten ist. Datenbanksysteme, die über diese Funktionalität verfügen, werden auch aktive Datenbanksysteme genannt.

Bleibt die Frage, warum wir uns in diesem Kapitel – anders als zum Beispiel in den vorangegangenen Kapiteln zum relationalen und objektrelationalen SQL – dazu entschlossen haben, auf die eigentliche Präsentation des SQL-Standards zu verzichten und stattdessen zwei Implementierungen vorzustellen. Zum einen ist es gerade bei Programmiersprachen nicht besonders interessant, wenn man die Anweisungen nicht auch in Programmen ausprobieren kann. Zum anderen sind die sehr jungen (seit 2005) gespeicherten Routinen von MySQL eine Implementierung, die sich sehr eng an SQL2003 orientiert, so dass man doch einen guten Einblick in den SQL-Standard bekommt, ohne auf praktische Übungen verzichten zu müssen. Und PL/SQL stellen wir vor, um einen Eindruck davon zu vermitteln, was alles in einem Datenbanksystem gemacht werden kann, wenn seit mehr als 20 Jahren (seit 1991) Benutzeranforderungen in einer Programmiersprache umgesetzt werden. Und wenn man dann bedenkt, dass der SQL-Standard erst 1996 seine „persistent stored modules“ herausgebracht hat, die mit dem 1999er-Standard nochmals überarbeitet und erweitert wurden (heute „SQL invoked routines“), dann findet man auch die Gründe, warum viele Hersteller mit ihren Programmiersprachen davon abweichen.

Die hier vorgestellten prozeduralen Sprachen und aktiven Regelkonzepte stellen nicht nur Erweiterungen des relationalen Modells dar. Sie können gleichermaßen im objektrelationalen Kontext eingesetzt werden. PL/SQL ist bei Oracle eine der Sprachen, mit denen Methoden für benutzerdefinierte Typen programmiert werden können (vgl. Abschnitt 6.1.9), ganz analog zum SQL-Standard, der dafür auch die gespeicherten Routinen vorsieht. Da es für die Trigger in SQL und bei Oracle unerheblich ist, ob das DML-Ereignis für eine relationale oder eine typisierte Tabelle ausgeführt wird, können aktive Regeln auch im objektrelationalen Kontext eingesetzt werden. Bei MySQL wird dies sicherlich noch realisiert, wenn erst einmal Methoden und Typtabellen implementiert sind.

Um Ihnen die bei MySQL „brandneuen“ Features wie Trigger und „Stored Routines“ vorstellen und erläutern zu können, haben wir mit der MySQL 5.1.11 Beta-Version gearbeitet. Alle Beispiele und Musterlösungen sind auf diesem System lauffähig. Für Oracle haben wir das Release 10g verwendet. <<

## Ziele

Nach Durcharbeiten dieses Kapitels und dem Lösen der Übungsaufgaben werden Sie

- sowohl in PL/SQL als auch in MySQL „gespeicherte Routinen“ programmieren und ausführen können,
- ein Verständnis entwickeln für das CURSOR-Konzept, das zur Lösung des „Impediment Match“ bei beiden Sprachen verwendet wird und grundlegend auch für andere Schnittstellen wie z.B. JDBC ist,
- das PL/SQL-Paket-Konzept anwenden können und einen Einblick in die umfangreiche Funktionsbibliothek von Oracle haben,
- aktive und passive Datenbanksysteme voneinander unterscheiden können und Verständnis für typische Anwendungsfälle für aktive Regeln haben,
- aktive Regelkonzepte in der Theorie, bei SQL, bei Oracle und MySQL kennen und in ihrer Leistungsfähigkeit beurteilen können,
- aktive Regeln programmieren können sowie
- die Unterschiede in den Ausführungsmodellen kennen sowie die Probleme für die Programmierung, die aus ihnen resultieren.

## 7.1 Die Datenbankprogrammiersprache Oracle-PL/SQL

PL/SQL ist eine Oracle-spezifische prozedurale Erweiterung von SQL, die auf ADA basiert. Sie stellt somit eine Möglichkeit dar, die mengenorientierten SQL-DML-Anweisungen Datensatz für Datensatz zu verarbeiten. PL/SQL wird im relationalen Kontext verwendet für im Datenbanksystem gespeicherte Routinen, im objektrelationalen für Methoden (vgl. Abschnitt 6.1.9) und im Kontext aktiver Regeln für die Triggeraktionen (vgl. Abschnitt 7.3). Wegen des Umfangs von PL/SQL ist es nicht möglich, die Sprache komplett zu beschreiben. Es soll lediglich ein Auszug dargestellt werden, der insbesondere für die Erstellung von Stored Procedures und Triggern notwendig ist.

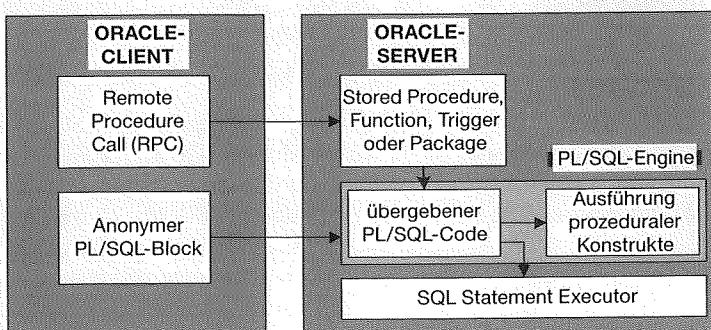


Abbildung 7.1: PL/SQL-Engine auf dem DB-Server

PL/SQL kann sowohl beim Server als auch beim Client ausgeführt werden. Findet die Ausführung auf dem Server statt, dann handelt es sich um in der Datenbank gespeicherte Prozeduren und Funktionen, die durch externe Aufrufe (RPC Remote Procedure Call) angestoßen werden oder über Datenbanktrigger.

PL/SQL-Aufrufe können alternativ auch beim Client verarbeitet werden. In diesem Fall wird der PL/SQL-Programmcode auf der Client-Seite bearbeitet und das Ergebnis dem Oracle-Datenbank-Server übergeben. Da wir uns auf die Anwendung von PL/SQL in Triggern und gespeicherten Routinen konzentrieren, betrachten wir nur das erste Ausführungsmodell.

### 7.1.1 Datentypen und andere Grundlagen

Die im Abschnitt 5.2.1 erläuterten SQL-Datentypen sind auch für PL/SQL implementiert. Eine Ausnahme ist der Datentyp BOOLEAN, der zwar unter SQL nicht realisiert ist, dafür aber unter PL/SQL. Analog zur dreiwertigen Logik in PL/SQL und SQL kann der Datentyp BOOLEAN drei logische Zustände annehmen: TRUE, FALSE, NULL/UNKNOWN. Problematisch ist die Verwendung dieses Datentyps nur als Ein- oder Aus- bzw. Rückgabewert für Funktionen, die direkt in einer SQL-Umgebung aufgerufen werden. Denn dort kann keine SQL-Variable mit entsprechendem Typ definiert werden. Statt der booleschen Parameter müssen dann Parameter vom Typ VARCHAR oder NUMBER spezifiziert werden. Ausgehend von diesen Basisdatentypen können vom Benutzer mittels der SUBTYPE-Deklaration beliebige abgeleitete Datentypen definiert werden. Sind Typkonvertierungen notwendig, so bieten sich analog zu SQL zwei Wege an: die explizite oder die implizite Konvertierung. Wenn die implizite Konvertierung nicht weiterhilft, kann man die explizite verwenden. Die Funktionen sind die gleichen, die wir bereits im Abschnitt 5.2.3 im Kontext von SQL-Funktionen vorgestellt haben.

Tabelle 7.1

#### Ein Ausschnitt aus den zusammengesetzten Datentypen<sup>1</sup>

Datentyp	Beschreibung
TYPE ...	Benutzerdefinierter Typ
TYPE ... IS VARRAY(...)	Ein Vektor fester Länge
TYPE ... IS RECORD(...)	Verbundtyp: Struktur eines Datensatzes mit unterschiedlichen Attributen
TYPE ... IS TABLE(...)	Eindimensionales (!) Array variabler Länge, bei dem auch ein Record als Datentyp in der Klammer eingetragen werden kann.
Tabellenname%ROWTYPE	Die Attributstruktur der Tabelle wird komplett übertragen.
Tabellenname.Spaltenname%TYPE	Die Variable wird entsprechend eines zuvor definierten Attributs einer Tabelle definiert.

<sup>1</sup> Eine vertiefende Erörterung des Themas finden Sie u.a. in [Oracle PL/SQL 2005], [Feuerstein et al. 2005] und [Urmann et al. 2005]

```
<Variablen-deklaration> ::= Variablename [CONSTANT] <Datentyp> [NOT NULL]
[ / DEFAULT | := ] <Ausdruck>;
```

Die optionale Eigenschaft CONSTANT legt fest, dass der Wert der Variablen nicht verändert werden kann. „Datentyp“ bezeichnet einen skalaren oder zusammengesetzten – auch benutzerdefinierten (vgl. Abschnitt 6.1.5) – Datentyp wie in der obigen Tabelle oder einen SQL-Datentyp. Variablen und Konstanten können in der Deklaration sofort mit „:=“ oder über die DEFAULT-Option initialisiert werden. Ohne Initialisierung ist ihr Zustand „leer“, was durch NULL symbolisiert wird. Eine Deklaration wird wie alle PL/SQL-Befehle mit einem Semikolon abgeschlossen. %TYPE- und %ROWTYPE-Deklarationen haben den Vorteil, dass bei Änderungen der Tabellendefinitionen automatisch bei der nächsten Ausführung des PL/SQL-Codes auch die Variablentypen aktualisiert werden.

Wie in anderen Programmiersprachen auch, können in PL/SQL an jeder beliebigen Stelle **Kommentare** untergebracht werden. Das doppelte Minuszeichen „--“ wird verwendet, um einzeilige Kommentare einzufügen und die beiden Zeichen /\* \*/ begrenzen einen mehrzeiligen Kommentar.

### Beispiel

```
Pruefe      BOOLEAN;
-- Dies ist ein einzeiliger Kommentar
Faktor      CONSTANT NUMBER(2) := 10;
Kundenname  Kunden.Nachname%TYPE DEFAULT 'kein Name';
Heute       DATE DEFAULT SYSDATE; -- bzw. ein Inline-Kommentar
Einzelteil  Teile%ROWTYPE;
/* und dies ist ein
   mehrzeiliger Kommentar */
ProdNr      Teile.TNr%TYPE;
TYPE Produkt_typ IS RECORD (Einzelteil Teile%ROWTYPE,
                           LetzterVerkauf DATE);
TYPE Feld IS TABLE OF INTEGER INDEX BY BINARY_INTEGER;
Artikelnummern  Feld;
Statuszahlen  Feld := Feld(1,2,3);
TYPE Student_t IS RECORD (Vorname  VARCHAR2(20),
                           Nachname VARCHAR2(20),
                           Gehalt    NUMBER(6,2));
```

### 7.1.1.1 PL/SQL-RECORD

Ein RECORD entspricht einem RECORD in einer 3GL-Sprache. Er hat ein oder mehrere Felder, die wieder einen skalaren, einen RECORD- oder einen anderen PL/SQL-Datentyp haben können. Wie obiges Beispiel zeigt, kann ein RECORD mittels %ROWTYPE und/oder einer Auflistung einzelner Spalten deklariert werden. Durch eine RECORD-Definition werden diese unterschiedlichen Datentypen als logische Einheit behandelt. Auf ein Record-Element wird über „recordname.elementname“ zugegriffen.

### 7.1.1.2 Index-by-Table

Index-by-Table, auch PL/SQL-Tabellen genannt, ähneln den Java- oder C-Arrays. Es werden der Tabellentyp und eine Variable dieses Typs deklariert.

```
TYPE Tab_Typname IS TABLE OF <Datentyp> INDEX BY BINARY_INTEGER;
Index_by_Tabellenname  Tab_Typname;
```

### Beispiel

```
TYPE Zahlen_Typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
Zahlen Zahlen_Typ;
BEGIN
  Zahlen(27) := 3;
END;
```

Elemente einer PL/SQL-Tabelle werden über einen Index referenziert. Außerdem gibt es noch vordefinierte Funktionen (COUNT, DELETE, FIRST, LAST, NEXT, PRIOR, EXISTS), mit denen man in Arrays navigieren oder Werte verändern kann. Benötigt man eine mehrspaltige PL/SQL-Tabelle, dann kann man als Datentyp in der TYPE-Deklaration auch einen RECORD verwenden. Die Wertzuweisung hat, im Gegensatz zu Java, einen Doppelpunkt vor dem Gleichheitszeichen.

```
<Wertzuweisung> ::= Variablename := <Ausdruck>;
```

Arithmetische, Vergleichs- und boolesche Operatoren sind hier die allseits bekannten, wie wir auch schon im Abschnitt 5.2.3 erläutert haben<sup>2</sup>. Die Typkonvertierungsfunktion CAST und die Konkatenation von Zeichenketten mittels des Operators „||“ oder der CONCAT-Funktion sind auch dort bereits vorgestellt worden.

Die Funktion „DBMS\_OUTPUT.PUT\_LINE(String)“ schreibt eine Zeile auf den Bildschirm, wenn und nur wenn vorher die Anweisung „SET SERVEROUTPUT ON“ ausgeführt bzw. wenn die Option „Environment/serveroutput“ auf „on“ umgeschaltet wurde. Ansonsten ist die Ausgabe trotz Funktionsausführung nicht zu sehen. Der „string“ kann auch mittels „||“ oder CONCAT aus verschiedenen Zeichenketten zusammengesetzt sein.

### 7.1.2 PL/SQL-Blöcke

Die kleinste Einheit von PL/SQL ist ein sogenannter PL/SQL-Block, der aus bis zu drei PL/SQL-Abschnitten bestehen kann.

Variablen- und Konstantendeklarationen haben wir bereits kennengelernt und die Cursor sowie die EXCEPTIONS werden wir in den Abschnitten 7.1.4 und 7.1.5 vorstellen. Die Definition von Unterprogrammen (subprocedure, subfunction) erfolgt in

<sup>2</sup> Eine vollständige Liste aller Operatoren und ihrer Funktionalität finden Sie in [ORACLE SQL 2005, Kap. 5], [Feuerstein et al. 2005, S. 208 ff., 262 ff.].

der gleichen Weise wie die Programmdefinition im Paketrumpf (vgl. Abschnitt 7.1.2), als CREATE PROCEDURE / FUNCTION-Anweisung, nur ohne das Schlüsselwort CREATE, beginnend also mit dem Schlüsselwort PROCEDURE bzw. FUNCTION. Jede PL/SQL-Anweisung, mit Ausnahme von DECLARE, BEGIN und EXCEPTION, endet mit einem Semikolon. Zulässige Anweisungen im Ausführungsteil sind hier alle PL/SQL-Befehle (vgl. Abschnitte 7.1.3 bis 7.1.5), die DML- sowie die Transaktionssteuerungsbefehle. Die Syntax der SELECT-Anfrage ist um die INTO-Klausel erweitert, damit die Anfragen in PL/SQL ausführbar sind (vgl. Abschnitt 7.1.4). Für die Ausführung von SQL-DDL-Anweisungen in PL/SQL wird Native Dynamic SQL (NDS) benötigt (vgl. Abschnitt 7.1.4).

Tabelle 7.2

PL-SQL-Abschnitte		
Abschnitt	Beschreibung	Vorhandensein
Deklarationsteil [DECLARE ...]	Enthält Deklarationen von benutzerdefinierten Datentypen, Variablen, Konstanten, CURSOR und benutzerdefinierte EXCEPTIONS, Unterprozeduren und -funktionen ...	optional
Ausführungsteil BEGIN ... END;	Enthält PL/SQL-Anweisungen und bestimmte SQL-Anweisungen wie INSERT, UPDATE, DELETE, SELECT, COMMIT, ROLLBACK	erforderlich
Fehlerbehandlungsteil [EXCEPTION ...]	Gibt an, welche Aktionen ausgeführt werden sollen, wenn im Ausführungsteil Fehler auftreten	optional

```
<PL/SQL-Block> ::=  
[ DECLARE  
  <Deklaration>; [ <Deklaration>; ]... ]  
BEGIN  
  <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]...  
[ EXCEPTION  
  <Fehlerbehandlung> ]  
END;
```

### 7.1.2.1 PL/SQL-Einheiten

Eine PL/SQL-Einheit enthält einen oder mehrere PL/SQL-Blöcke. PL/SQL-Blöcke können in PL/SQL-Einheiten geschachtelt werden. Werden Blöcke geschachtelt, so ist auf die Gültigkeit der Variablen und Konstanten zu achten. Auf Variablen kann im eigenen Block und in allen Unterblöcken zugegriffen werden. In übergeordneten oder Blöcken gleicher Ebene sind die Variablen nicht gültig. Für das nachfolgende Beispiel gilt u.a., dass die Variablen von Block 1 in allen anderen Blöcken gültig sind, die von Block 3 jedoch nur in Block 3 selbst.

### Beispiel

```
SET SERVEROUTPUT ON;  
  
BEGIN  
  DBMS_OUTPUT.PUT_LINE('Start 1. Block');  
  BEGIN  
    DBMS_OUTPUT.PUT_LINE('..Jetzt bin ich im 2. Block');  
    BEGIN  
      DBMS_OUTPUT.PUT_LINE('...und jetzt im 3. Block');  
    END;  
    BEGIN  
      DBMS_OUTPUT.PUT_LINE('....und jetzt im 4. Block');  
    END;  
    DBMS_OUTPUT.PUT_LINE('..und zurück im 2. Block');  
  END;  
  DBMS_OUTPUT.PUT_LINE('Ende 1. Block');  
/  
SHOW ERRORS
```

Die beiden Anweisungen Schrägstrich „//“ (Abkürzung für den SQL\*PLUS-Befehl RUN) und SHOW ERRORS werden bei der Verwendung der SQL\*PLUS-Oberfläche von Oracle benötigt. Bei anderen Entwicklungsoberflächen wie z.B. den TOAD<sup>3</sup> sind diese Programmabschlüsse nicht notwendig. Der Schrägstrich „//“ beendet die Eingabe einer PL/SQL-Anweisung (hier alterner BEGIN-END-Block) und veranlasst ihre Ausführung. Treten dabei Kompilierungsfehler auf, so werden sie mit SHOW ERRORS angezeigt. Auch die erste Anweisung SET SERVEROUTPUT ON wird nur bei SQL\*PLUS benötigt. Sie sorgt dafür, dass die Ausführungen der DBMS\_OUTPUT.PUT\_LINE-Funktion sichtbare Bildschirmausgaben erzeugen (vgl. Abschnitt 7.1.1).

### 7.1.2.2 Hauptkategorien von PL/SQL-Blöcken

Auf dem Datenbankserver werden hinsichtlich der gespeicherten Routinen vier Hauptkategorien von PL/SQL-Blöcken unterschieden (vgl. Tabelle 7.3).

Die anonymen Blöcke sowie die gespeicherten Routinen unterscheiden sich im Aufbau, wie aus der Grafik ersichtlich wird:

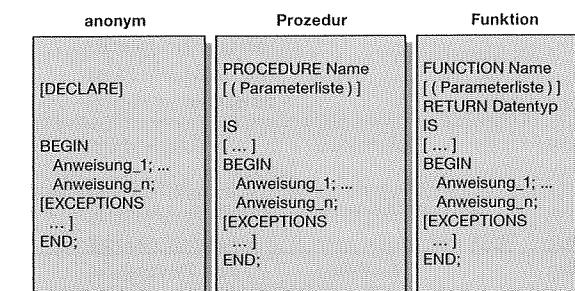


Abbildung 7.2: Struktur der PL/SQL-Blocktypen: anonym, Prozedur, Funktion

<sup>3</sup> TOAD ist ein Datenbankentwicklungstool, welches in der Praxis recht weite Verbreitung gefunden hat. Es wird von Quest Software entwickelt, <http://www.quest.com/de>, 05.03.06

Tabelle 7.3

Unterscheidung der vier Hauptkategorien von PL/SQL-Blöcken	
Blocktyp	Beschreibung
Anonymer Block	Unbenannter PL/SQL-Block, der in einer Anwendung (Prozedur, Funktion ...) eingebettet ist oder interaktiv eingegeben wird.
Stored Routine	Benannter PL/SQL-Block, der Parameter haben kann und als Prozedur oder Funktion definiert ist. Er wird im Datenbanksystem gespeichert und auf dem Server von der PL/SQL-Engine ausgeführt.
PACKAGE	Benannter PL/SQL-Block, der logisch verwandte Prozeduren und Funktionen, Deklarationen etc. zu einer Bibliothek zusammenfasst.
Datenbanktrigger	PL/SQL-Block, der zu einem definierten Ereignis automatisch vom Datenbankmanagementsystem aktiviert und ausgeführt wird. Da die Datenbanktrigger eigentlich ein SQL-Konzept sind, werden sie nicht hier erläutert, sondern im Abschnitt 7.3.

### 7.1.2.3 Gespeicherte Routinen

Zunächst beschäftigen wir uns mit den Prozeduren und anschließend mit den Funktionen, die sich nur geringfügig unterscheiden. Wenn sowohl Prozeduren als auch Funktionen gemeint sind, verwenden wir den Begriff Routine.

PL/SQL-Prozeduren sind benannte PL/SQL-Blöcke, die Parameter aufnehmen und wieder zurückgeben können. Es werden mehrere SQL- und PL/SQL-Anweisungen zu einer Einheit zusammengefasst.

```
<CREATE PROCEDURE Anweisung> ::= 
  CREATE [OR REPLACE] PROCEDURE Name
  [ ( <Parameterdefinition> [ , <Parameterdefinition> ]... ) ]
  IS
  [ <Deklaration>; [ <Deklaration>; ]... ]
  BEGIN
  <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]...
  [ EXCEPTION
  <Fehlerbehandlung> ]
  END;

<Parameterdefinition> ::= [ IN | OUT | IN OUT ] <Datentyp>
[ / DEFAULT Wert ] | / := Wert / ]
```

Es gibt drei Typen der Parameterübergabe: IN, OUT und IN OUT. IN ist der DEFAULT-Wert, wenn der Parametertyp nicht angegeben wird. Die Parameter werden zwar mit ihrem Datentyp spezifiziert, aber ohne Längenangabe. Die ergibt sich bei der Programm-ausführung. Die Bedeutung der Typen wird in Tabelle 7.4 erläutert.

Beispiel

Schreiben Sie eine Prozedur „Mitteln“, an die zwei Zahlen x und y übergeben werden und die als Ergebnis  $(x + y) / 2$  liefert (vgl. MySQL, Abschnitt 7.2.2)!

```
CREATE OR REPLACE PROCEDURE Mitteln (x IN NUMBER, y IN NUMBER)
IS
  Ergebnis NUMBER(20);
BEGIN
  Ergebnis:= (x + y) / 2;
  DBMS_OUTPUT.PUT_LINE('Ergebnis = ' || Ergebnis);
END;
/
SHOW ERRORS;
SET SERVEROUTPUT ON;
EXECUTE Mitteln(7,11);
```

Tabelle 7.4

Die drei Typen der Parameterübergabe

	IN	OUT	IN OUT
Funktion	Wertübergabe beim Aufruf in das Programm hinein	Wertübergabe aus dem Programm heraus zurück an das aufrufende Objekt	Wertübergabe in initialisierter Form an die Prozedur und Rückgabe eines veränderten Werts an das aufrufende Objekt
Verhalten	verhält sich wie eine Konstante innerhalb des Programms	verhält sich wie eine nicht initialisierte Variable, die nur einen Wert aufnehmen und an das aufrufende Objekt zurückgeben kann	verhält sich wie eine initialisierte Variable, die einen Wert aufnehmen und an das aufrufende Objekt zurückgeben kann
DEFAULT-Wert	möglich; Wenn kein Wert übergeben wird beim Aufruf, wird dieser DEFAULT-Wert im Programm verarbeitet	nicht möglich	nicht möglich
Wertzuweisung im Programm	nicht möglich	erforderlich	möglich
Aufruparameter	kann eine initialisierte Variable, Konstante, ein Ausdruck sein	muss eine Variable sein, wegen dem Rückgabewert	muss eine Variable sein, wegen dem Rückgabewert

Der SQL\*PLUS-Befehl SHOW ERRORS zeigt die Fehler des letzten Kompiliervorgangs an. Das EXECUTE-Kommando führt in SQL\*PLUS eine gespeicherte Prozedur aus und kann mit EXEC abgekürzt werden. Die Anweisung DROP PROCEDURE Prozedur-

name; löscht eine Prozedur wieder aus der Datenbank. Mit dem Befehl GRANT EXECUTE ON Prozedurname TO Benutzername; können Sie Ausführungsrechte an einzelne Benutzer vergeben, mit dem Befehl GRANT EXECUTE ON Prozedurname TO PUBLIC; räumen Sie diese Ausführungsrechte allen Benutzern ein.

Funktionen unterscheiden sich von Prozeduren im Wesentlichen darin, dass sie immer zumindest einen Rückgabewert haben und zwar in Form des RETURN-Parameters. Darüber hinaus können weitere Werte mittels der OUT- und IN OUT-Parameter zurückgegeben werden. Beim Aufruf einer Funktion werden Parameterwerte wie auch bei den Prozeduren mittels der IN- und IN OUT-Parameter übergeben. In der Regel werden Funktionen aufgrund ihres Rückgabeparameters für die Berechnung bzw. Ermittlung eines Werts programmiert.

```
<CREATE FUNCTION Anweisung> ::=  
  CREATE [ OR REPLACE ] FUNCTION Name  
    [ ( <Parameterdefinition> [ , <Parameterdefinition> ]... ) ]  
    RETURN <Datentyp>  
    IS  
      [ <Deklaration>; [ <Deklaration>: ]... ]  
    BEGIN  
      <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung> ]...  
    [ EXCEPTION  
      <Fehlerbehandlung> ]  
    END;  
  
<RETURN Anweisung> ::= RETURN [ <Variable> | <Konstante> | <Ausdruck> ];
```

Die Syntax der CREATE FUNCTION-Anweisung unterscheidet sich von der CREATE PROCEDURE-Anweisung nur in der zusätzlichen RETURN-Klausel. Als Datentyp wird dort ein zulässiger PL/SQL-Datentyp angegeben. Daraus resultieren nun ein paar Abweichungen gegenüber der Prozedurausführung. Während eine Prozedurausführung bei der END-Anweisung endet, muss bei der Funktion zumindest eine RETURN-Anweisung im Aktionsteil programmiert sein, deren Aufgabe die Rückgabe des Parameters und das Beenden des Programms an genau dieser Stelle ist. Mit der Anweisung DROP FUNCTION Funktionname; wird eine Funktion wieder gelöscht.

### Beispiel

Schreiben Sie eine Funktion „Gehaltssumme“, die das Gesamtgehalt über alle Angestellten ermittelt (vgl. MySQL, Abschnitt 7.2.2)!

```
CREATE OR REPLACE FUNCTION Func_Gehaltssumme  
RETURN NUMBER IS  
  V_Summe NUMBER;  
BEGIN  
  SELECT SUM(Gehalt) INTO V_Summe FROM Angestellte;  
  RETURN V_Summe;  
END;  
/  
SHOW ERRORS
```

### 7.1.2.4 Aufruf von Funktionen in SQL\*PLUS

Der Rückgabeparameter der Funktionen erfordert eine andere Aufrufart als die Prozeduren. Es gibt drei Möglichkeiten, Funktionen aufzurufen.

**1. Möglichkeit: in SQL-Anweisungen** Funktionen, wie hier Func\_Gehaltssumme, können grundsätzlich in jedem SQL-Anfrage- und Manipulationsbefehl aufgerufen und ausgeführt werden. Sie können überall dort aufgerufen werden, wo SINGLE ROW-Funktionen aufrufbar sind (vgl. Abschnitt 5.2.3), z.B.:

```
SELECT Func_Gehaltssumme FROM Angestellte;
```

Da in der SELECT-Anweisung die Funktion für jeden Datensatz der Ergebnismenge aufgerufen wird, wird im obigen Beispiel die Funktion für jeden Datensatz der Angestelltentabelle ausgeführt. Will man eine solche Funktion garantiert nur einmal ausführen, so kann man bei Oracle die Pseudotabelle „DUAL“ nutzen. DUAL enthält nur einen Datensatz und somit werden SELECT-Anfragen auch nur einmal ausgeführt.

```
SELECT Func_Gehaltssumme FROM DUAL;
```

**2. Möglichkeit: in PL/SQL-Anweisungen** Alternativ kann man die Funktion auch mittels eines geeigneten PL/SQL-Befehls ausführen. Geeignete Anweisungen sind z.B. Zuweisungen oder als Parameter eines Prozedur- oder Funktionsaufrufs. Eine andere Möglichkeit ist die Verwendung der Anzeigefunktion PUT\_LINE aus dem Bibliotheks-paket DBMS\_OUTPUT (vgl. Abschnitt 7.1.2.2), womit dann die Funktion ausgeführt und gleichzeitig das Ergebnis angezeigt wird. Denken Sie aber bei SQL\*PLUS daran, den SERVEROUTPUT-Parameter auf ON zu setzen (vgl. Abschnitt 7.1.1).

```
BEGIN  
  DBMS_OUTPUT.PUT_LINE('Die Gehaltssumme ist: '||Func_Gehaltssumme);  
END;  
/  
DECLARE  
  V_Summe  NUMBER(10);  
BEGIN  
  V_Summe := Func_Gehaltssumme;  
  DBMS_OUTPUT.PUT_LINE ('Die Summe der Gehälter beträgt: '||V_Summe);  
END;  
/
```

**3. Möglichkeit: mit SQL\*PLUS-Variablen** Dieser Aufruf funktioniert nur unter SQL\*PLUS. Eine Sessionvariable wird definiert, mit dem der Funktionswert über die EXECUTE-Anweisung gefüllt und über den PRINT-Befehl auf dem Bildschirm ausgegeben, z.B.:

```
VARIABLE Gehaltssumme NUMBER;  
EXECUTE :Gehaltssumme:= Func_Gehaltssumme();  
PRINT Gehaltssumme;
```

### 7.1.2.5 Pakete

Pakete (PACKAGES) sind Datenbankobjekte, mit denen logisch in Verbindung stehende Programmkonstrukte, wie

- Prozeduren,
- Funktionen,
- CURSOR,
- Variablen und Konstanten sowie
- EXCEPTIONS,

zu einer Einheit zusammengefasst werden. Sie haben damit eine gewisse Ordnungsfunktion. Ihre Abspeicherung erfolgt in kompakter Form in der Datenbank.

Ein Paket besteht aus einem Spezifikationsteil und einem Rumpf, in dem die spezifizierten Objekte mit Programmcode definiert sind.

```
<CREATE PACKAGE Anweisung> ::=  
  CREATE [ OR REPLACE ] PACKAGE Paketname [ IS | AS ]  
    <Spezifikation Kopf>; [ <Spezifikation Kopf>; ]...  
  END [ Paketname ];
```

Die Spezifikation, auch Header genannt, enthält in der <Spezifikation Kopf> die Deklaration der Variablen, Konstanten, Prozeduren etc., die zu dem Paket gehören. Nicht alle Objekte, die im Paketrumpf definiert sind, müssen auch in der Spezifikation auftauchen, Grund dafür ist das Konzept der privaten und öffentlichen Paketobjekte (s.u.).

#### Beispiel

```
CREATE OR REPLACE PACKAGE Math_Pack AS  
  PROCEDURE Mitteln (x IN NUMBER, y IN NUMBER);  
  PROCEDURE Root (x IN NUMBER, y IN NUMBER);  
  Ergebnis NUMBER;  
END Math_Pack;
```

```
<CREATE PACKAGE BODY Anweisung> ::=  
  CREATE [ OR REPLACE ] PACKAGE BODY Paketname [ IS | AS ]  
    <Spezifikation Rumpf>; [ <Spezifikation Rumpf>; ]...  
  BEGIN  
    <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]...  
  END [ Paketname ];
```

Der Paketrumpf, auch Body genannt, enthält in der <Spezifikation Rumpf> die Deklaration der lokalen Variablen, Konstanten etc. sowie die Programmdefinition aller Prozeduren und Funktionen, die zu dem Paket gehören. Er kann auch über einen eigenen Ausführungsteil (zwischen BEGIN und END) verfügen, der ausgeführt wird, wenn zum ersten Mal in einer Sitzung irgendeine Komponente des Pakets aufgerufen wird. Im Allgemeinen werden in diesem Ausführungsteil Initialisierungsaufgaben programmiert.

#### Beispiel

```
CREATE OR REPLACE PACKAGE BODY Math_Pack AS  
  PROCEDURE Mitteln (x IN NUMBER, y IN NUMBER) IS  
  BEGIN  
    Ergebnis := (x + y) / 2;  
    DBMS_OUTPUT.PUT_LINE('Ergebnis = ' || Ergebnis);  
  END Mitteln;  
  PROCEDURE Root (x IN NUMBER, y IN NUMBER) IS  
  BEGIN  
    Ergebnis := SQRT (x * y);  
    DBMS_OUTPUT.PUT_LINE('Ergebnis = ' || Ergebnis);  
  END Root;  
  BEGIN  
    Ergebnis := 0;  
  END;  
/  
SHOW ERRORS  
-- ausgeführt werden die Prozeduren mit der SQL-PLUS-Anweisung  
-- EXECUTE Paketname.prozedurname  
EXECUTE Math_Pack.Mitteln(7,9);  
EXECUTE Math_Pack.Root(2,8);
```

**Private und öffentliche Paketobjekte** Je nach Platzierung eines Paketobjekts, wie Prozedur, Funktion, Variable, Konstante, Cursor etc., ist es entweder öffentlich (public) oder privat (private):

- Alle Objekte, die in der Paketspezifikation deklariert sind, sind öffentlich (public) und können damit von außerhalb des Pakets mit Paketname.Objektname aufgerufen werden.
- Alle Objekte, die nur im Paketrumpf definiert sind, sind privat (private) und können damit nur innerhalb des Pakets von den Paketroutinen aufgerufen werden.

Damit bieten Pakete die wichtige Funktion der Kapselung von Funktionalität. Alle Objekte, die nicht von außerhalb verwendet werden sollen, werden im Paket „versteckt“, indem ihre Spezifikation nicht im Kopfteil vorkommt.

Beim obigen Paket Math\_Pack sind die Prozeduren Mitteln und Root sowie die Variable Ergebnis öffentlich. Private Objekte gibt es nicht. Im Ausführungsteil des Paketrumpfs wird die Variable Ergebnis beim ersten Ansprechen des Pakets in der laufenden Sitzung mit 0 initialisiert.

#### Beispiel

```
CREATE PACKAGE XYZ  
IS  
  Const_4711 CONSTANT NUMBER(9) := 4711;  
  PROCEDURE Proc_b (p1 IN NUMBER);  
END;
```

Die Konstante Const\_4711 sowie die Prozedur Proc\_b sind eine öffentliche Konstante bzw. Methode. Sie können von außerhalb des Pakets XYZ in beliebigen PL/SQL-Blöcken sowie in SQL-Anfrage- und Modifikationsanweisungen mit der Notation XYZ.const\_4711 bzw. XYZ.proc\_b aufgerufen werden. Wie der Paketrumpf zeigt, können Objekte auch im eigenen Rumpf ausgeführt werden.

```
CREATE PACKAGE BODY XYZ
IS
  V_d VARCHAR2(20);
  PROCEDURE Proc_a (p1 NUMBER) IS ... END;
  PROCEDURE Proc_b (p1 NUMBER) IS ... END;
  PROCEDURE Proc_c (par_1 IN OUT NUMBER) IS ...
BEGIN
  Proc_b(v_d);
  Proc_a(const_4711);
END;
END;
```

Die Variable V\_d sowie die Prozeduren Proc\_a und Proc\_c sind private Methoden, die außerhalb des Pakets nicht aufgerufen werden können. Bei der Definition von Proc\_c wird gezeigt, dass private wie öffentliche Objekte innerhalb des Pakets verwendet werden können.

### 7.1.2.6 Overloading

Beim Overloading können Paketroutinen gleich heißen: Gleichnamige Prozeduren oder Funktionen werden durch eine unterschiedliche Anzahl, eine unterschiedliche Reihenfolge von Parametern oder durch unterschiedliche Datentypen der Parameter unterschieden. Diese Funktionalität ist nicht auf Paketprogramme beschränkt, sondern für alle Subprogramme möglich, die im Deklarationsteil irgendeines PL/SQL-Blocks definiert sind. Wir zeigen diese Funktionalität beispielhaft für Paketroutinen.

#### Beispiel

```
CREATE OR REPLACE PACKAGE XYZ_overload
IS
  /* zulässig, da erste Deklaration*/
  PROCEDURE a (p1 NUMBER);

  /* zulässig, da eine unterschiedliche Anzahl an Parametern*/
  PROCEDURE a (p1 NUMBER, p2 VARCHAR2);

  /* zulässig, da eine unterschiedliche Reihenfolge der Parameter*/
  PROCEDURE a (p1 VARCHAR2, p2 NUMBER);

  /* zulässig, da unterschiedliche Datentypen*/
  PROCEDURE a (p1 DATE);

  /* NICHT zulässig, da weder unterschiedliche Anzahl noch
   unterschiedliche Datentypen zur 1. Definition*/
  PROCEDURE a (p3 NUMBER);

END;
/
SHOW ERRORS
```

```
CREATE OR REPLACE PACKAGE BODY XYZ_overload
IS
  -- zulässig, da erste Deklaration
  PROCEDURE a (p1 NUMBER) IS
    BEGIN DBMS_OUTPUT.PUT_LINE(p1); END;

  -- zulässig, da eine unterschiedliche Anzahl an Parametern
  PROCEDURE a (p1 NUMBER, p2 VARCHAR2) IS
    BEGIN DBMS_OUTPUT.PUT_LINE(p1||' '||p2); END;

  -- zulässig, da eine unterschiedliche Reihenfolge der Parameter
  PROCEDURE a (p1 VARCHAR2, p2 NUMBER) IS
    BEGIN DBMS_OUTPUT.PUT_LINE(p1||' '||p2); END;

  -- zulässig, da unterschiedliche Datentypen
  PROCEDURE a (p1 DATE) IS
    BEGIN DBMS_OUTPUT.PUT_LINE(p1); END;

  /* NICHT zulässig, da weder unterschiedliche Anzahl noch
   unterschiedliche Datentypen zur 1. Definition*/
  PROCEDURE a (p3 NUMBER) IS
    BEGIN DBMS_OUTPUT.PUT_LINE(p3); END;

END;
/
SHOW ERRORS
EXECUTE XYZ_overload.a(1);
EXECUTE XYZ_overload.a(1, 'X');
EXECUTE XYZ_overload.a('Y', 1);
EXECUTE XYZ_overload.a(SYSDATE);
```

Der Fehler „PLS-00307: too many declarations of 'A' match this call“ tritt nicht beim Kompilieren auf, sondern erst beim Zugriff auf die Prozeduren, die sich vom DBMS nicht eindeutig identifizieren lassen. Die hier aufgeführten Ausführungsbefehle funktionieren nur, wenn die fehlerhafte Deklaration von Prozedur a aus Kopf und Rumpf entfernt wird.

### 7.1.2.7 Oracle-Bibliothek

Die sehr umfangreiche Oracle-Bibliothek besteht aus einer ganzen Reihe mitgelieferter Pakete, von denen hier einige aufgeführt werden sollen.<sup>4</sup>

**DBMS\_STANDARD** Dieses Paket enthält eigentlich die gesamte PL/SQL-Programmumgebung, wie Datentypen, EXCEPTIONS, Subprogramme (AVG, MOD, RAISE, RETURN, WHEN ...) etc. Es ist insofern ein untypisches Paket, als dass seine Objekte nicht nur über die übliche Namenskonvention Paketname.Objektname angesprochen werden können, sondern auch einfach nur mit dem Objektnamen aufgerufen werden können. Der Befehl

<sup>4</sup> Für eine vollständige Übersicht verweisen wir auf die Oracle-Originalliteratur [Oracle PL/SQL 2005], [Oracle Packages 2005] oder die Bücher [Feuerstein et al. 2005] und [Urman et al. 2005].

`RAISE_APPLICATION_ERROR` aus diesem Paket zur Ausgabe eigener Fehlermeldungen und zum Erzeugen eines Programmabbruchs wird beim Fehler-Handling im Abschnitt 7.1.5 erläutert.

Wir werden den Befehl auch noch bei der Triggerprogrammierung im Abschnitt 7.3 oft verwenden.

**DBMS\_OUTPUT** bietet die Möglichkeit, Nachrichten auf dem Bildschirm z.B. beim Debuggen anzuzeigen. Wir nutzen hauptsächlich die Funktion `DBMS_OUTPUT.PUT_LINE(Text)` zur Ausgabe in SQL\*PLUS und `DBMS_OUT-PUT.ENABLE(Buffergröße)` zum Einstellen der Größe des Bildschirmbuffers.

**DBMS\_SQL** Mit Hilfe des dynamischen SQL können SQL-Anweisungen ausgeführt werden, die erst zur Laufzeit erzeugt werden oder die sonst nicht in PL/SQL ausgeführt werden können. Dieses Paket war lange Zeit die Standardlösung für diese Problemstellung, wird jetzt aber in seiner Funktionalität immer mehr abgelöst durch das komfortablere Native Dynamic SQL (NDS, vgl. Abschnitt 7.1.5).

**DBMS\_JOB** Die Routinen dieses Pakets unterstützen die Ausführungsplanung von Routinen mit periodischer Wiederholung. Ein häufiges Einsatzgebiet ist die automatisierte Durchführung von administrativen Tätigkeiten. Das Paket stellt auch die Schnittstelle zur Verwaltung der Job-Warteschlange bereit.

**DBMS\_PIPES** und **DBMS\_AQ** DBMS\_PIPES stellen einen Pipe-Service zur Verfügung, der zwischen Sessions (Sitzungen) Kommunikation auf der Basis von Nachrichten ermöglicht. Der neuere Kommunikationsservice heißt Advanced Queuing (AQ) und bietet Kommunikationsdienstleistungen an, die weit über die Funktionalität des alten Pakets DBMS\_PIPES hinausgehen.

**DBMS\_OBFUSCATION\_TOOLKIT** Mittels dieser Routinen können Daten auf der Basis des Data Encryption Standards (DES)<sup>5</sup> oder des Triple DES (3DES) ver- und entschlüsselt werden. Der DES-Algorithmus arbeitet mit 64-Bit-Schlüsseln. Eine stärkere Verschlüsselung ist bei 3DES mit 128-Bit- oder 192-Bit-Schlüsseln zu erreichen.

**UTL\_FILE** bietet die Möglichkeit, in Dateien des Betriebssystems zu schreiben oder aus ihnen zu lesen.

**HTP** und **HTF** Diese beiden Pakete enthalten Routinen, um HTML-Seiten aus der Datenbank zu generieren.

**DBMS\_LOB** spielt eine zentrale Rolle bei dem Umgang mit den Multimediadatentypen und enthält Funktionen und Prozeduren zum Bearbeiten von Bild-, Audio- und Videodaten sowie großen Texten, die als „large objects“ in der Datenbank abgespeichert werden. Für BLOBs und CLOBs gibt es Lese- und Schreibroutinen, für BFILEs nur Leseoperationen. Die Funktionen aus dem DBMS\_LOB-Paket können nicht interaktiv aufgerufen werden, sondern nur über Schnittstellen (JDBC, PL/SQL). Sie beinhalten unter anderem folgende Methoden:

<sup>5</sup> Der Data Encryption Standard (DES) ist mehr als 20 Jahre praktisch im Einsatz und auch bekannt als Data Encryption Algorithmus (DEA) des American National Standards Institute (ANSI) bzw. als DEA-1 des International Standards Organization (ISO). Es ist geplant, dass DES durch den neuen Advanced Encryption Standard (AES) ersetzt wird.

Tabelle 7.5

Funktionen und Prozeduren des Pakets DBMS_LOB	
Name	Beschreibung
APPEND	Hängt die Inhalte eines LOB-Werts an einen anderen LOB-Wert an
CLOSE	Schließt einen zuvor geöffneten internen oder externen LOB
COMPARE	Vergleicht zwei LOB-Werte
CONVERTOCLOB	Liest Character-Daten aus einer Quell-CLOB- oder NCLOB-Instanz, konvertiert die Character-Daten in die vorgegebenen Zeichen und hängt die konvertierten Daten im binären Format an eine Ziel-BLOB-Instanz an
COPY	Kopiert den gesamten LOB oder bestimmte Teile in ein Ziel-LOB
CREATETEMPORARY	Legt einen temporären BLOB- oder CLOB-Index Wert an
ERASE	Löscht den LOB ganz oder teilweise
FILECLOSE	Schließt die Datei
FILECLOSEALL	Schließt alle zuvor geöffneten Dateien
FILEEXISTS	Prüft, ob die Datei vorhanden ist
FILEGETNAME	Liest den Dateinamen und den Verzeichnisnamen
FILEISOPEN	Prüft, ob die Datei mithilfe des Eingabe-BFILE-Locator geöffnet wurde
FILEOPEN	Öffnet eine Datei
FREETEMPORARY	Gibt das temporäre BLOB oder CLOB im standardmäßigen temporären Tablespace des Benutzers frei
GET_STORAGE_LIMIT	Liefert das Speicherlimit für LOBS in Ihrer Datenbankkonfiguration
GETCHUNKSIZE	Liefert den Speicherplatz des LOB-Wertes
GETLENGTH	Holt die Länge des LOB-Wertes ab
INSTR	Liefert die Position eines Musters im LOB
ISOPEN	Prüft, ob das LOB bereits mithilfe des Eingabe-Locator geöffnet wurde
ISTEMPORARY	Prüft, ob der Locator auf einen temporären LOB zeigt
LOADBLOBFROMFILE	Lädt BFILE-Daten in einen internen BLOB
LOADCLOBFROMFILE	Lädt BFILE-Daten in einen internen CLOB
LOADFROMFILE	Lädt BFILE-Daten in einen internen LOB
OPEN	Öffnet ein LOB (intern, extern oder temporär) im angegebenen Modus
READ	Liest die Daten des LOB ab der vorgegebenen Position
SUBSTR	Liefert Teile des LOB-Werks ab der vorgegebenen Position
TRIM	Kürzt den LOB-Wert auf die vorgegebene Länge
WRITE	Schreibt die Daten ab der vorgegebenen Position in das LOB
WRITERAPPEND	Schreibt einen Puffer an das Ende des LOB

**Beispiel****zur Verwendung der LOB-Funktionen**

Mit dieser Anweisung wird eine Tabelle mit LOB-Spalten angelegt:

```
CREATE TABLE Mitarbeiter (
    Name VARCHAR(30),
    Bild BLOB,
    Bilddatei BFILE,
    Bewerbung CLOB,
    Bewerbungsdatei BFILE);
```

Mit der nächsten Anweisung wird ein Verzeichnis angelegt, in dem die Daten für die BLOB-Dateien zu finden sind. Das Verzeichnis muss auf dem gleichen Server liegen, auf dem auch der Datenbankserver installiert ist.

```
CREATE DIRECTORY BLOB_DIRECTORY AS
    '/oracle/OraHome1/blob_verzeichnis/';
INSERT INTO Mitarbeiter VALUES ('Meier',
    EMPTY_BLOB(),
    BFILENAME('BLOB_DIRECTORY', 'Meier.gif'),
    EMPTY_CLOB(),
    BFILENAME('BLOB_DIRECTORY', 'Meier.txt'))
);
```

Dieser INSERT-Befehl initialisiert die BLOB-Spalten BILD und Bewerbung mittels der Funktionen EMPTY\_BLOB() bzw. EMPTY\_CLOB() und lädt mit der Funktion BFILENAME einen Zeiger auf ein BFILE in die entsprechenden Spalten.

```
CREATE OR REPLACE PROCEDURE setBewerbung (InputName IN VARCHAR2)
IS Textdaten CLOB;
    Datei BFILE;
BEGIN
    SELECT Bewerbung, Bewerbungsdatei
    INTO Textdaten, Datei
    FROM Mitarbeiter
    WHERE Name = InputName FOR UPDATE;
    DBMS_LOB.FILEOPEN(Datei, DBMS_LOB.FILE_READONLY);
    DBMS_LOB.LOADFROMFILE(Textdaten, Datei, DBMS_LOB.GETLENGTH(Datei));
    DBMS_LOB.FILECLOSE(Datei);
    UPDATE Mitarbeiter
    SET Bewerbung = Textdaten
    WHERE Name = InputName;
END;
/
EXECUTE setBewerbung('Meier');
```

Die Prozedur SetBewerbung lädt die BFILE-Daten der Bewerbungsdatei in eine lokale Variable Datei vom Typ Bfile, die zum Namen des Mitarbeiters passt, der der Prozedur setBewerbung als Übergabeparameter übergeben wird. SELECT FOR UPDATE ist notwendig, um die Daten für die weitere Bearbeitung vorübergehend zu sperren. Anschließend wird auf diesen Mitarbeiter (hier: Name = 'Meier') ein UPDATE durchgeführt, das die CLOB-Spalte mit einem Wert füllt. Die Prozedur wird über EXECUTE ausgeführt.

Mit der nun folgenden Prozedur showBewerbung wird die CLOB-Spalte angezeigt:

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE PROCEDURE showBewerbung (InputName IN VARCHAR2)
IS Puffer VARCHAR(60);
    Textdaten CLOB;
    Textlaenge DECIMAL;
    Position DECIMAL;
BEGIN
    Textlaenge := 60;
    Position := 1;
    SELECT Bewerbung INTO Textdaten
    FROM Mitarbeiter
    WHERE Name = InputName;
    DBMS_LOB.READ(Textdaten, Textlaenge, Position, Puffer);
    DBMS_OUTPUT.PUT_LINE(Puffer);
END;
/
EXECUTE ShowBewerbung('Meier');
```

**7.1.3 Ablaufsteuerung und Kontrollstrukturen**

Wie in jeder anderen Programmiersprache auch, gibt es in PL/SQL verschiedene Konstrukte, um Ablaufsteuerung und Kontrollstrukturen zu unterstützen. Auch hier können wir nur die gängigsten besprechen.

In PL/SQL gibt es:

- Sequenzen von PL/SQL-Anweisungen, die durch Semikolon getrennt werden
- Bedingte Verzweigungen mit IF und CASE
- Verschiedene Schleifen: Basisschleife ohne Bedingung LOOP, FOR-, WHILE- und CURSOR-Schleifen
- Eine EXIT-Anweisung zum Verlassen von Schleifen

**7.1.3.1 Bedingte Verzweigungen**

```
<IF Anweisung> ::= 
    IF <Bedingung> THEN
        <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]...
        [ ELSIF <Bedingung> THEN
            <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]... ]...
        [ ELSE
            <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]... ]
    END IF;
```

Die Bedingungen sind boolesche Bedingungen beliebiger Komplexität und unterscheiden sich nur geringfügig von den Suchbedingungen, die wir im Abschnitt 5.5 für die WHERE-Klausel eingeführt haben. Anders als in der WHERE-Klausel können die Spaltennamen der Tabellen nicht verwendet werden, da sich die IF-Anweisung nicht auf eine Tabelle bezieht. Stattdessen werden PL/SQL-Variablen benutzt. Aber auch hier liegt eine dreiwertige Logik zugrunde. Das Ergebnis der Auswertung kann TRUE,

FALSE oder NULL (unbekannt) sein. Es können auch Aufrufe von PL/SQL-Funktionen sein, die einen booleschen Rückgabewert haben. Wenn die Bedingung TRUE ist, werden die Anweisungen im THEN-Zweig ausgeführt und anschließend wird zum END IF gesprungen. Ist sie FALSE oder NULL, wird der nächste Zweig abgearbeitet. Dieser nächste Zweig kann ein END IF sein, so dass ohne Ausführen von irgendwelchen Anweisungen die IF-Anweisung beendet wird. Er kann ein ELSE sein, dessen Anweisungen in dem Fall ausgeführt werden, oder er kann ein ELSIF sein, bei dem wieder eine Bedingung zu prüfen ist. Es sind beliebig viele ELSIF-Klauseln erlaubt. Wenn es weitere ELSIF-Zweige gibt, so werden deren Bedingungen, sobald eine zu TRUE ausgewertet wurde, nicht mehr geprüft. Nur eine ELSE-Klausel ist zulässig. Auf die Erläuterung der CASE-Anweisung verzichten wir hier, da sie funktionell mit einer IF-ELSIF-Anweisung simuliert werden kann. Ihr Vorteil liegt in besseren Laufzeiten, wenn die gleiche Bedingung in mehreren ELSIF-Zweigen wiederholt ausgewertet werden muss.

### Beispiel<sup>6</sup>

```
IF V_Anzahl = 0 THEN V_Ausgabe := 'A';
ELSIF V_Anzahl = 1 THEN V_Ausgabe := 'B';
ELSIF V_Anzahl = 2 THEN V_Ausgabe := 'C';
ELSE V_Ausgabe := 'Weder A noch B noch C';
END IF;
```

### 7.1.3.2 Schleifen

Die einfachste Schleife besteht aus Wiederholungen zwischen den Schlüsselwörtern LOOP und END LOOP. Eine der Anweisungen innerhalb der Schleife muss eine EXIT-Anweisung sein. Da die Basisschleife ohne irgendeine Bedingung formuliert wird, fehlt sonst ein Terminierungskriterium und die Schleife wird endlos durchlaufen. Mit der EXIT-Anweisung können auch die anderen Schleifentypen vorzeitig verlassen werden. Für die Bedingung hier gelten die gleichen Aussagen wie bei der IF-Anweisung.

```
<LOOP-Schleife> ::= 
  LOOP
    <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]...
  END LOOP;

<EXIT Anweisung> ::= EXIT [ WHEN <Bedingung> ];
```

### Beispiel<sup>7</sup>

```
V_zahler:= 0;
LOOP
  V_zahler:= V_zahler + 1;
  INSERT INTO Zahler_Tabelle VALUES (V_zahler);
  EXIT WHEN V_zahler = 10;
END LOOP;
```

<sup>6</sup> vgl. MySQL, Abschnitt 7.2.3  
<sup>7</sup> vgl. MySQL, Abschnitt 7.2.3

Als Zählerschleife gibt es die FOR-Schleife.

```
<FOR-Schleife> ::= 
  FOR Indexvariable IN [REVERSE] Untergrenze .. Obergrenze LOOP
    <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]...
  END LOOP;
```

Der Index durchläuft einen Bereich, der die Werte der unteren und oberen Grenze mit einschließt. Bei den gegebenenfalls negativen Grenzwerten muss es sich nicht um ein numerisches Literal, eine Variable oder Konstante handeln, es können auch Ausdrücke sein, die numerische Werte zurückliefern. Bei der Schleifenausführung werden solche Ausdrücke aber nur beim ersten Auswerten der Schleifenbedingung ausgewertet, dann sind die Grenzen fix für alle Wiederholungen. Die Indexvariable ist eine ganzzahlige Variable, die implizit deklariert wird. Sie wird immer um den Wert 1 inkrementiert oder dekrementiert. Außerhalb der Schleife ist sie nicht zugreifbar und innerhalb kann sie nur als Konstante verwendet werden. Die Option REVERSE gibt an, dass der Index bei jedem Schleifendurchlauf erniedrigt wird.

### Beispiel

Schreiben Sie eine Prozedur, die zu einer natürlichen Zahl die Fakultät  $1 \cdot 2 \cdot 3 \cdot 4 \cdots \cdot n$  berechnet!

```
CREATE OR REPLACE PROCEDURE Fakultaet (n NUMBER) IS
  Ergebnis NUMBER;
BEGIN
  Ergebnis:= 1;
  FOR i IN 1..n LOOP
    Ergebnis := Ergebnis * i;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE
    ('Die Fakultät von '||n|| ' ist: '|| Ergebnis);
END;
/
SHOW ERRORS
EXECUTE Fakultaet(4);
```

Neben der Zählerschleife gibt es noch eine kopfgesteuerte WHILE-Schleife, die so lange wiederholt wird, bis die kontrollierende Bedingung nicht mehr „TRUE“ ist. Auch hier ist wieder eine beliebige Bedingung zulässig, wie wir sie für die IF-Anweisung vorgestellt haben.

```
<WHILE-Schleife> ::= 
  WHILE <Bedingung> LOOP
    <SQL- und PL/SQL-Anweisung>; [ <SQL- und PL/SQL-Anweisung>; ]...
  END LOOP;
```

**Beispiel**

In einer Prozedur soll für den übergebenen Wert die Fakultät berechnet werden (vgl. MySQL, Abschnitt 7.2.3).

```
CREATE OR REPLACE PROCEDURE Fakultaet (n NUMBER) IS
  Ergebnis NUMBER := 1;
  i      NUMBER := 1;
BEGIN
  WHILE i < n+1 LOOP
    Ergebnis := Ergebnis * i;
    i := i+1;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('Die Fakultät von '||n||' ist: ' ||Ergebnis);
END;
/
SHOW ERRORS
EXECUTE Fakultaet (3);
```

### 7.1.4 Datenbankzugriffe innerhalb von PL/SQL

Ein großer Vorteil der Programmierung mit PL/SQL ist die unproblematische Verwendung von SQL-Anweisungen. Damit orientiert sich Oracle im Konzept und in der Syntax genau am SQL-Standard. Ohne irgendwelche Schnittstellenkonstrukte werden die SQL-Anweisungen genauso wie die PL/SQL-Anweisungen geschrieben.

#### 7.1.4.1 SQL-Anweisungen

In PL/SQL-Blöcken können Sie:

- SELECT-, INSERT-, UPDATE- oder DELETE-Anweisungen einbauen,
- Transaktionen mit COMMIT oder ROLLBACK kontrollieren sowie
- das Ergebnis von SQL-Anweisungen mit Cursorn weiterverarbeiten.

In PL/SQL-Blöcken sind nicht erlaubt:

- DDL-Anweisungen wie CREATE, ALTER, DROP und
- die Vergabe von Benutzerrechten mit GRANT oder REVOKE.

Während die INSERT-, UPDATE- und DELETE-Anweisungen exakt den SQL-Anweisungen entsprechen, gibt es bei SELECT-Anweisungen einen Unterschied, da das Ergebnis eines SELECT in einer Variablen gespeichert werden muss.

```
<SELECT INTO Anweisung> ::=  
  <SELECT Klausel>  
  INTO / Variablename [ , Variablename ]... | Recordname  
  <FROM Klausel>  
  [ <WHERE Klausel> ]  
  [ <GROUP BY Klausel> ]  
  [ <HAVING Klausel> ]  
  [ <ORDER BY Klausel> ];
```

Mit der INTO-Komponente wird das Ergebnis der SELECT-Anweisung in eine PL/SQL-Variable oder einen PL/SQL-RECORD geschrieben. Die Namen der Variablen sollten sich von den Spaltennamen der Tabellen unterscheiden, um Verwechslungen auszuschließen. Die Datentypen der Ausgabevariablen der INTO-Komponente müssen mit den Datentypen der Datenbankspalten der SELECT-Komponente übereinstimmen, ebenso die Reihenfolge mit korrespondierenden Typen, denn die Zuweisung zwischen SQL-Spalten und PL/SQL-Variable erfolgt nicht über die Namen, sondern über die Position. Bei der Deklaration der INTO-Variablen und -RECORDS ist die Verwendung der %TYPE- bzw. %ROWTYPE-Syntax sinnvoll, um Datentypkompatibilität zu gewährleisten.

Dieser SELECT INTO-Befehl hat eine wichtige Besonderheit bei der Ausführung: Er wird nur dann korrekt ausgeführt, wenn genau ein Datensatz aus der Datenbasis selektiert wird. Nur in diesem Fall wird die Programmausführung „ganz normal“ bei der nachfolgenden Anweisung fortgesetzt. Besteht die Ergebnismenge aus mehreren Datensätzen oder ist sie leer, dann tritt ein TOO\_MANY\_ROWS- bzw. ein NO\_DATA\_FOUND-Ausführungsfehler auf. Bei einem solchen Fehler bricht der SELECT INTO-Befehl ab und die Programmausführung springt an das Ende des Ausführungsteils. Ist die entsprechende EXCEPTION definiert, dann wird die dort programmierte Fehleraktion ausgeführt. Sind die passenden EXCEPTIONS nicht spezifiziert, bricht das Programm fehlerhaft ab (vgl. Abschnitt 7.1.5). Aufgrund dieser Besonderheit bei der Ausführung sollte ein SELECT INTO grundsätzlich zusammen mit diesen beiden EXCEPTIONS programmiert werden. Der SELECT INTO-Befehl ist zum einen gut zu verwenden, wenn nur ein Ergebnisdatensatz zu erwarten ist. Das ist z.B. der Fall, wenn Aggregatfunktionen wie MIN, MAX, COUNT, SUM, AVG ... ohne Gruppierungen (ohne GROUP BY-Klausel) ausgewertet werden oder wenn mit einem Gleichheitsvergleich („=“) auf den Primärschlüssel oder andere Eindeutigkeitsschlüssel (UNIQUE KEYs) zugegriffen wird. Zum anderen wird der SELECT INTO-Befehl aber auch angewendet, wenn es um Tests geht. Je nachdem, wie das Ergebnis ist, ob ein, kein oder mehrere Datensätze gefunden wurden, wird hinter der SELECT INTO-Anweisung oder bei den EXCEPTIONS mit NO\_DATA\_FOUND bzw. mit TOO\_MANY\_ROWS fortgefahrene. Hier kann es dann sein, dass aufgrund der Anwendungslogik diese Ausführungsfehler gar keine Fehler im Sinne der Anwendung sind (keine semantischen Fehler) und dass das Programm in den oder einer der EXCEPTIONS ganz normal weiterläuft.

**Beispiel**

#### eine kleine Torstatistik mit SELECT INTO-Befehl

Es soll im WM-Schema Rollo eine Prozedur geschrieben werden, die eine kleine Torstatistik führt und die maximale, minimale und durchschnittliche Torminute für alle Tore anzeigen, bei denen die Zeit bekannt ist. Die Torminute 999 steht für Tore, die durch Elfmeter erzielt wurden. Diese Sätze sollen nicht berücksichtigt werden (vgl. MySQL, Abschnitt 7.2.4).

```
CREATE OR REPLACE PROCEDURE Kleine_Torstatistik
IS
  V_min      NUMBER(3);
  V_max      NUMBER(3);
  V_avg      NUMBER(5,2);
```



```

BEGIN
  SELECT MIN(Minute), MAX(Minute), AVG(Minute)
  INTO   V_min, V_max, V_avg
  FROM   Tore
  WHERE  Minute <999;
  DBMS_OUTPUT.PUT_LINE('Maximale Torminute: ' || V_max);
  DBMS_OUTPUT.PUT_LINE('Minimale Torminute: ' || V_min);
  DBMS_OUTPUT.PUT_LINE('Durchschnittliche Torminute: ' || V_avg);
END;
/
SHOW ERRORS
EXEC Kleine_Torstatistik;

```

In diesem Beispiel kann auf die beiden Standard-EXCEPTIONS verzichtet werden, da es sich hierbei um ungruppierte Aggregationen handelt, die auf jeden Fall nur ein Ergebnis liefern. Der einzige Ausnahmefall tritt ein, wenn keine Datensätze in der Toretabelle gespeichert sind. Ein Fall, der sicherlich nicht sehr häufig vorkommt. Aber selbst dann werden die Aggregationen eben über die leere Menge ausgeführt und liefern NULL als Ergebnis zurück – also wird keine EXCEPTION ausgelöst.

### Beispiel

#### eine kleine Spieltagsstatistik mit SELECT INTO-Befehl



Schreiben Sie eine Prozedur, der die Nummer eines Spieltags übergeben wird. Für diese übergebene Nummer wird zuerst getestet, ob es ein vorhandener Spieltag ist. Anschließend werden aus den Informationen der Ergebnisspalte die Summe der Tore und die Anzahl der unentschiedenen Spiele für diesen Tag ermittelt (vgl. PL/SQL, Abschnitt 7.1.5).

```

CREATE OR REPLACE PROCEDURE Spieltagstatistik (P_Tag IN NUMBER) IS
  V_Torsumme      NUMBER;
  V_unentschieden NUMBER;
  V_Tag           NUMBER;

BEGIN
  -- Test, ob p_tag ein Spieltag ist
  SELECT DISTINCT Spieltag INTO V_Tag
  FROM Spiele
  WHERE Spieltag = P_Tag;

  SELECT SUM(SUBSTR(Ergebnis,1,INSTR(Ergebnis,':')-1)
             + SUBSTR(Ergebnis,INSTR(Ergebnis,':')+1,LENGTH(Ergebnis)))
  INTO V_Torsumme
  FROM Spiele
  WHERE Spieltag = P_Tag;
  DBMS_OUTPUT.PUT_LINE( CONCAT('Summe der Tore: ',V_Torsumme));

  SELECT COUNT(*) INTO V_unentschieden
  FROM Spiele
  WHERE spieltag = p_tag
  AND  SUBSTR(Ergebnis,1,INSTR(Ergebnis,':')-1) =
        SUBSTR(Ergebnis,INSTR(Ergebnis,':')+1,LENGTH(Ergebnis));
  DBMS_OUTPUT.PUT_LINE( CONCAT
    ('Anzahl unentschiedener Spiele an dem Tag: ',V_Unentschieden));

```

```

EXCEPTION
  WHEN TOO_MANY_ROWS THEN DBMS_OUTPUT.PUT_LINE
    ('Zuviele Sätze gefunden!!!');
  WHEN NO_DATA_FOUND THEN DBMS_OUTPUT.PUT_LINE('Kein Satz gefunden in
                                                der Spieltagstatistik!!!');

END;
/
SHOW ERRORS

CALL Spieltagstatistik (1);
CALL Spieltagstatistik (2);
-- Diesen Tag gibt es nicht
CALL Spieltagstatistik (23);

```

In Vorgriff auf die im Abschnitt 7.1.5 ausführlich erläuterte Fehlerbehandlung mittels EXCEPTION haben wir hier das Beispiel schon mal mit den beiden Fehlern NO\_DATA\_FOUND und TOO\_MANY\_ROWS komplettiert, damit Sie erst gar nicht den Eindruck bekommen, man könne – außer bei wohl begründeten Ausnahmen wie in diesem Fall – auf eine oder gar beide EXCEPTIONS verzichten. Beide Beispiele sollen Ihnen schon mal ein Gefühl dafür vermitteln, dass bei jedem SELECT INTO eigene Überlegungen angestellt werden müssen, ob und welche Fehlerfälle auftreten und damit behandelt werden müssen. Wenn Sie die Option DISTINCT bei der Testanfrage weglassen, können Sie den Fehler TOO\_MANY\_ROWS auslösen.

### 7.1.4.2 Das CURSOR-Konzept

Vor dem Hintergrund, dass man als Programmierer ja auch mal Mengen von Datensätzen verarbeiten möchte und nicht immer nur einen Satz, stellt die Eigenschaft der SELECT INTO-Anweisung, nicht mehrere Ergebnismengen zurückzugeben zu können, ein echtes Manöver dar. Will man aber mit einer prozeduralen Programmiersprache eine Datensatzmenge verarbeiten, dann stößt man auf das „Impediment Mismatch“-Problem (siehe ► Abbildung 7.3). Der Grund ist, dass das DBMS eine Datensatzmenge unbekannter Größe zurück liefert und dass die Variablen im Programm nur einen Datensatz aufnehmen können. Um also eine Menge von Datensätzen aus der Datenbasis im Programm verarbeiten zu können, muss ein sogenannter CURSOR mit einer SELECT-Anfrage deklariert werden (siehe ► Abbildung 7.3). Für diesen Cursor wird dann ein Zwischenspeicherbereich reserviert. Das Programm schickt die Anfrage an das Datenbankmanagementsystem, das sie auswertet und dann die Ergebnismenge in den reservierten Zwischenspeicherbereich schreibt. Das PL/SQL-Programm greift auf diesen Zwischenspeicher zu, indem es die dortige Ergebnismenge in einer Schleife Datensatz für Datensatz abarbeitet. Dieses CURSOR-Konzept findet sich in seinen Grundzügen und unter anderem Namen bei eigentlich allen prozeduralen Sprachen wieder, die über eine SQL-Schnittstelle verfügen (vgl. RESULT SET (Abschnitt 6.2.1) bei JDBC und ITERATOR bei SQLJ (Abschnitt 6.2.2)).

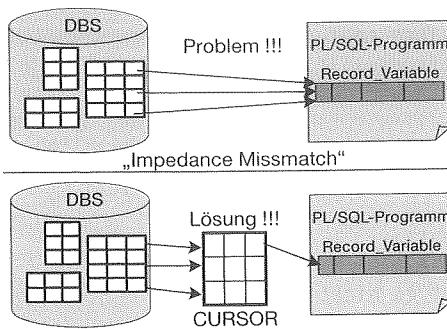


Abbildung 7.3: Grundidee eines CURSORs

In PL/SQL unterscheidet man je nach Art der Deklaration explizite und implizite CURSOR.

Tabelle 7.6

Implizite und explizite Cursor	
Typ	Beschreibung
Implizite CURSOR	Werden implizit von PL/SQL für jede SELECT-Anweisung (SELECT INTO, CURSOR FOR-Schleifen,...) definiert und hier nicht weiter betrachtet. Sie sind eine komfortable Kurzschreibweise der expliziten CURSOR.
Explizite CURSOR	Werden vom Programmierer deklariert und dienen zur Weiterverarbeitung der Ergebnismengen von SELECT-Anweisungen. Da hier für jeden Verarbeitungsschritt explizit die Befehle angegeben werden, wollen wir diese Syntax verwenden, damit für Sie das dahinter liegende Konzept offensichtlicher wird. Auf das Konzept der REF-Cursor gehen wir hier auch nicht weiter ein. <sup>8</sup>

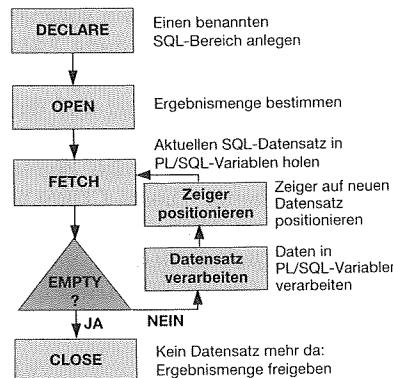


Abbildung 7.4: Ausführen eines CURSORs

<sup>8</sup> REF-Cursor basieren auf Cursor-Variablen, die mehr Flexibilität ermöglichen, da sie nicht an eine bestimmte Anfrage fest gebunden sind. Nähere Informationen finden Sie in [Oracle PL/SQL 2005, Kap. 6].

**Arbeitsweise von CURSORN** ► Abbildung 7.4 gibt einen Überblick über die Schritte, die bei der Ausführung eines CURSOR gemacht werden.

- **DECLARE:** Als erstes wird der CURSOR im Deklarationsteil des PL/SQL-Blocks definiert, das heißt, er bekommt einen Namen und eine SELECT-Anweisung, die beliebig komplex sein kann. Eine INTO-Klausel ist weder erlaubt noch notwendig. Damit verbunden ist intern das Anlegen eines benannten Zwischenspeicherbereichs für die Aufnahme der Ergebnismenge. Alle weiteren Schritte werden im Ausführungsteil durchgeführt.
- **OPEN:** Beim Öffnen eines CURSOR wird die zugehörige SELECT-Anfrage an das Datenbankmanagementsystem geschickt und dort ausgewertet. Von dort wird die Ergebnismenge in den benannten Speicherbereich geladen, was heißt, dass während der gesamten Zeit der CURSOR-Verarbeitung die Ergebnisdatenmenge unverändert bleibt. Nach der OPEN-Anweisung parallel von anderen Transaktionen durchgeföhrte Manipulationen haben keinen Einfluss mehr auf die Ergebnisdaten, die ja bereits als Kopie im Zwischenspeicherbereich vorliegen.
- **FETCH:** In PL/SQL ist ein Zeiger (CURSOR) nach dem Öffnen sofort auf den ersten Datensatz dieser Ergebnismenge positioniert, so dass mit einer FETCH-Anweisung dieser Satz aus dem benannten Speicherbereich in die lokalen PL/SQL-Variablen hinein ausgelesen werden kann. Als zusätzliche Funktion positioniert die FETCH-Anweisung den Zeiger auch noch auf den nächsten Datensatz im Zwischenspeicher<sup>9</sup>.
- **EMPTY:** Bevor der gerade eingelesene Datensatz verarbeitet wird, muss geprüft werden, ob überhaupt Daten ausgelesen werden konnten oder ob die Variablen leer sind, weil kein Datensatz mehr im Zwischenspeicherbereich war.
- **Verarbeiten und Positionieren:** War der letzte FETCH hingegen erfolgreich in dem Sinne, dass ein Datensatz in die lokalen Variablen geladen werden konnte, so können diese Daten nun verarbeitet werden. Anschließend wird der Zeiger auf den nächsten Datensatz im Zwischenspeicherbereich positioniert und die Schleife erneut durchlaufen.
- **CLOSE:** War der letzte FETCH nicht erfolgreich, d.h., sind die lokalen Variablen leer, dann heißt das, dass alle Datensätze abgearbeitet sind und die Schleife verlassen werden kann. Beim Schließen des CURSOR wird der damit verbundene Zwischenspeicherbereich wieder freigegeben.

```
<CURSOR Deklaration> ::= CURSOR Cursorname IS <SELECT Anweisung>;
```

### Beispiel

```
CURSOR Ang_Cursor IS SELECT Ang_Nr, Abt_Nr, Nachname, Gehalt
                      FROM Angestellte
                      ORDER BY Abt_Nr;
```

<sup>9</sup> Andere, vergleichbare Programmierkonstrukte, wie das ResultSet von JDBC (Abschnitt 6.2.2) bieten flexiblere Navigationsmöglichkeiten an.

```
<OPEN Anweisung> ::= OPEN Cursorname;
<FETCH Anweisung> ::=  
  FETCH Cursorname  
  INTO [ Variablename [, Variablename ]... | Recordname];
```

Mit der **FETCH**-Anweisung werden die Werte der aktuellen Zeile in die Ausgabevariablen oder einen Ausgabe-Record gelesen. Sie stehen somit einer weiteren internen Verarbeitung zur Verfügung. (Record-)Variablen und Spalten der Tabelle müssen die gleichen Datentypen haben.

### Beispiel

```
FETCH Ang_Cursor INTO V_Ang_Nr, V_Abt_Nr, V_Nachname, V_Gehalt;
```

Wenn Sie mehrere Zeilen aus einem expliziten CURSOR auslesen wollen, definieren Sie eine Schleife, die bei jedem Durchlauf einen **FETCH** ausführt. Wenn alle Zeilen abgearbeitet wurden, wird die Eigenschaft **%NOTFOUND** des CURSORS vom Datenbankmanagementsystem auf TRUE gesetzt und die Schleife kann mit **EXIT** verlassen werden. Mit der **CLOSE**-Anweisung wird dann der CURSOR geschlossen.

```
<CLOSE Anweisung> ::= CLOSE Cursorname;
```

**Attribute von expliziten CURSORN** Der Status eines CURSORS kann abgefragt werden, indem Sie dem Attributnamen den CURSOR-Namen voranstellen:

Tabelle 7.7

CURSOR-Attribute	
CURSOR-Attribut	Beschreibung
<b>%ISOPEN</b>	Boolesches Attribut, das TRUE ist, wenn der CURSOR geöffnet ist
<b>%NOTFOUND</b>	Boolesches Attribut, das TRUE ist, wenn die letzte <b>FETCH</b> -Anweisung keine Zeile mehr liefert
<b>%FOUND</b>	Gegenteil von <b>%NOTFOUND</b>
<b>%ROWCOUNT</b>	Gesamtanzahl der bisher gelesenen Zeilen; leider gibt es keine Möglichkeit, zu erfahren, wie groß die Ergebnismenge überhaupt ist.

### Beispiel 1

#### eines CURSORS

Geben Sie die ersten fünf Angestellten (in alphabetischer Reihenfolge) auf dem Bildschirm aus (vgl. MySQL, Abschnitt 7.2.4)!

```
CREATE OR REPLACE PROCEDURE Ang_Fuenf IS
  V_Abt_Nr      NUMBER;
  V_Nachname    VARCHAR2(20);
  CURSOR Ang_Cur IS
    SELECT Abt_Nr, Nachname
    FROM Angestellte
    ORDER BY Nachname;
  BEGIN
    OPEN Ang_Cur;
    LOOP
      FETCH Ang_Cur INTO V_Abt_Nr, V_Nachname;
      EXIT WHEN Ang_Cur%NOTFOUND OR Ang_Cur%ROWCOUNT > 5;
      DBMS_OUTPUT.PUT_LINE('Abt: ' || V_Abt_Nr || ' | ' ||
                           'Name: ' || V_Nachname);
    END LOOP;
    CLOSE Ang_Cur;
  END;
/
SHOW ERRORS
EXEC Ang_Fuenf;
```

### Beispiel 2

#### eines CURSORS

Alle Angestellten werden abteilungsweise auf dem Bildschirm ausgegeben. Die Ausgabe wird abgeschlossen mit der Angabe der Gehaltssumme aller Angestellten sowie dem durchschnittlichen Gehalt aller Angestellten (vgl. MySQL, Abschnitt 7.2.4).

```
CREATE OR REPLACE PROCEDURE Angestellte_ausgeben
IS
  V_Nachname    VARCHAR2(20);
  V_Gehalt      NUMBER;
  V_Abt_Nr      NUMBER;
  V_Summe       NUMBER := 0;
  V_Avg         NUMBER := 0;
  C_Trenn       CONSTANT VARCHAR2(4):= ',';
  CURSOR Ang_Cursor IS
    SELECT Abt_Nr, Nachname, Gehalt
    FROM Angestellte
    ORDER BY Abt_Nr;
  BEGIN
    DBMS_OUTPUT.ENABLE(20000);
    OPEN Ang_Cursor;
    FETCH Ang_Cursor INTO V_Abt_Nr, V_Nachname, V_Gehalt;
    WHILE Ang_Cursor%FOUND LOOP
      DBMS_OUTPUT.PUT_LINE('Abteilung: ' || V_Abt_Nr || C_Trenn ||
                           'Name: ' || V_Nachname || C_Trenn ||
                           'Gehalt: ' || TO_CHAR(V_Gehalt, '99999'));
      FETCH Ang_Cursor INTO V_Abt_Nr, V_Nachname, V_Gehalt;
    END LOOP;
    CLOSE Ang_Cursor;
```

```

SELECT SUM(Gehalt), AVG(Gehalt)
  INTO V_Summe, V_Avg
   FROM Angestellte;
DBMS_OUTPUT.PUT_LINE('Die Gehaltssumme aller Angestellten beträgt:
                      ||TO_CHAR(V_Summe,'999999')
                      ||' und das durchschnittliche Gehalt:
                      ||TO_CHAR(V_Avg,'999999'));
END;
/
SHOW ERRORS
EXECUTE Angestellte_ausgeben;

```

### 7.1.4.3 Dynamisches SQL mit Native Dynamic SQL (NDS)

Wie zu Beginn des Abschnitts 7.1.5 dargestellt, kann man in PL/SQL nur SQL-Manipulationsanweisungen, aber

- keine DDL-Anweisungen wie CREATE, ALTER, DROP etc.,
  - keine DCL wie GRANT, REVOKE etc. und
  - keine SESSION CONTROL-Anweisungen wie ALTER SESSION

verarbeiten. Einen Ausweg aus dieser Situation bietet dynamisches SQL unter PL/SQL mit NDS, dem Native Dynamic SQL. Die auszuführende SQL-Anweisung wird ohne abschließendes Semikolon als Zeichenkette oder in einer Textvariablen dem Befehl EXECUTE IMMEDIATE übergeben, der diesen Text ungeprüft zur Ausführung an das Datenbankmanagementsystem sendet. Der Befehl EXECUTE IMMEDIATE selbst wird aber mit einem Semikolon abgeschlossen.

```
<EXECUTE IMMEDIATE Anweisung> ::=  
  EXECUTE IMMEDIATE [ <SQL-Zeichenkette> | Textvariable ]  
    [ INTO [ Variablenname [, Variablenname ]... | Recordname ] ]  
    [ USING [ IN | OUT | IN OUT ] Parametername  
      [ , [ IN | OUT | IN OUT ] Parametername ]... ];
```

Die auszuführende SQL-Anweisung oder ein alterer PL/SQL-Block werden als einfache Hochkomma gesetzte Zeichenkette ohne abschließendes Semikolon entweder direkt angegeben oder zuvor einer Textvariablen zugewiesen. Für die Dateneingabe bei auszuführenden Anfragen ist die INTO-Klausel zuständig. Funktionalität und Restriktionen entsprechen denen der SELECT INTO-Anweisung. In der USING-Klausel sind es PL/SQL-Variablen oder Konstanten, deren Werte an die Hostvariablen in der SQL-Anweisung übergeben werden. Die Zuordnung erfolgt gemäß der Reihenfolge in der USING-Klausel.

Dynamisches SQL hat die Eigenschaft, SQL-Anweisungen erst zur Laufzeit zusammenzubauen und zu komplizieren (parsen). Dies hat den Nachteil, dass Syntaxfehler bei der Anweisung im SQL-String auch erst zur Laufzeit erkannt werden. Von Vorteil ist diese Eigenschaft, wenn man mehr Generalisierung und Flexibilität bei der Programmierung braucht und die Syntax der Anweisungen erst zur Laufzeit festlegen möchte. Man hat somit die Chance, mit einer EXECUTE IMMEDIATE-Anweisung unterschiedliche SQL-Anweisungen auszuführen, deren genaue Syntax zur Entwicklungszeit noch gar nicht bekannt ist, sondern sich erst zur Laufzeit z.B. durch interaktive Benutzereingaben bestimmen.

gaben ergibt. Eine sehr häufige Anwendung aus der Praxis sind wechselnde SELECT- und WHERE-Klauseln bei Anfragen<sup>10</sup>.

### **Beispiel**

Hier wird NDS genutzt, um in einem anonymen Block einen Index im Datenbankschema Rollo für die Tabelle Spiele über die beiden Mannschaftsspalten zu definieren (vgl. MySQL, Abschnitt 7.2.4).



```

DECLARE
  V_ddl_Anweisung  VARCHAR2(200);
BEGIN
  V_ddl_Anweisung := 'CREATE INDEX Spiele_Mannschaften_idx
                      ON Spiele(Mannschaft_1, Mannschaft_2)';
  EXECUTE IMMEDIATE V_ddl_Anweisung;
END;
/
SELECT * FROM User_Indexes;
-- Es geht auch ohne die PL/SQL-Textvariable:
DROP INDEX Spiele_Mannschaften_idx;

BEGIN
  EXECUTE IMMEDIATE 'CREATE INDEX Spiele_Mannschaften_idx
                      ON Spiele(Mannschaft_1, Mannschaft_2)';
END;
/
SHOW ERRORS
SELECT * FROM User_Indexes;

```

### Beispiel

Es wird eine SELECT-Anweisung ausgeführt und die Spalten des Ergebnisdatensatzes werden in zwei PL/SQL-Variablen geschrieben. Da über die Primärschlüsselkolonne auf Gleichheit selektiert wird, ist gewährleistet, dass es nur einen Ergebnisdatensatz gibt.

```

DECLARE
  V_Mannschaft_1    Spiele.Mannschaft_1%TYPE;
  V_Mannschaft_2    Spiele.Mannschaft_2%TYPE;
  V_Where_Text      VARCHAR2(200) := 'Spiel_Id = 1';
BEGIN
  EXECUTE IMMEDIATE 'SELECT Mannschaft_1, Mannschaft_2
                      FROM Spiele WHERE '||V_Where_Text
                      INTO V_Mannschaft_1, V_Mannschaft_2;
  DBMS_OUTPUT.PUT_LINE
    ('Im Spiel 1 spielten: '||V_Mannschaft_1||' gegen '
     ||V_Mannschaft_2);
END;
/
SHOW ERRORS

```

10 JDBC setzt wie PL/SQL dynamisches SQL um: Mit den beiden Methoden `executeQuery` und `executeUpdate` der Statement-Klasse (vgl. Abschnitt 6.2.2) werden auch zur Laufzeit zusammengestellte SQL-Anweisungen an die Datenbank geschickt.

### 7.1.5 Fehlerbehandlung

Fehlerbehandlung zur Laufzeit heißt in PL/SQL „EXCEPTION Handling“. Eine EXCEPTION ist das Eintreten eines an dieser Stelle nicht vorgesehenen Ereignisses, nicht zu verwechseln mit einem syntaktischen Fehler, der vom Parser zurückgewiesen wird. Der Fehlerbehandlungsteil gehört zu den optionalen Bestandteilen eines PL/SQL-Blocks.

```
<Fehlerbehandlung> ::=  
  EXCEPTION  
    WHEN Exceptionname [ OR Exceptionname ]...  
      THEN <SQL- und PL/SQL-Anweisung>;  
      [ <SQL- und PL/SQL-Anweisung>; ]...  
    [ WHEN Exceptionname [ OR Exceptionname ]...  
      THEN <SQL- und PL/SQL-Anweisung>;  
      [ <SQL- und PL/SQL-Anweisung>; ]... ]...  
    WHEN OTHERS THEN <SQL- und PL/SQL-Anweisung>;  
      [ <SQL- und PL/SQL-Anweisung>; ]...];
```

Eine Fehlerbehandlung besteht also aus einer Reihe von Anweisungen, die ausgeführt werden, wenn ein bestimmter Fehler auftritt. Tritt ein Fehler auf, so wird das Programm an der Stelle abgebrochen und es wird an das Ende des Ausführungsteils gesprungen. Nun gibt es im Wesentlichen zwei Möglichkeiten:

- Ist für diesen Fehler eine EXCEPTION programmiert, dann wird die EXCEPTION abgearbeitet. Es werden die vorgegebenen Fehleraktionen ausgeführt. Da nun aus Sicht der PL/SQL-Engine der Fehler bearbeitet wurde, wird die Prozedur ohne Fehler beendet und die ursprüngliche Fehlermeldung wird nicht auf dem Bildschirm angezeigt. Hier ist es also wichtig, dass man eine sinnvolle und vor allem aussagekräftige Fehlerreaktion vorgibt, damit nicht wichtige Fehlerinformationen verloren gehen.
- Wird der Fehler nicht durch eine EXCEPTION abgefangen, so bricht das Programm insgesamt fehlerhaft ab und die zugehörige Oracle-Fehlermeldung wird auf dem Bildschirm angezeigt.

**Tabelle 7.8**

Die drei verschiedenen EXCEPTION-Typen		
EXCEPTION-Typ	Beschreibung	Hinweise
Vordefinierter Oracle-Fehler	Einer von 21 häufig vorkommenden Fehlern	Nicht deklarieren; sie werden automatisch vom Oracle-Server auslöst
Nicht vordefinierter Oracle-Fehler	Ein anderer der auch so vielen Oracle-Standardfehler	Deklarieren, d.h. der Fehler hat einen Namen; sie werden automatisch vom Oracle-Server ausgelöst. Zum Behandeln des Fehlers den Namen dann in der WHEN-Klausel verwenden.
Benutzerdefinierter Fehler	Ein Programmzustand, den der Programmierer als falsch definiert	Im Deklarationsteil deklarieren und explizit im Programm auslösen. Mit diesem Thema werden wir uns hier nicht weiter beschäftigen.

**Tabelle 7.9**

### Einige vordefinierte EXCEPTIONS<sup>11</sup>

EXCEPTION-Name	Oracle-Fehler-Nr.	Beschreibung
DUP_VAL_ON_INDEX	ORA_00001	Versuch, einen doppelten Wert in eine Spalte einzutragen, die einen eindeutigen Index hat; Einfügen von Duplikaten in eine Tabelle
INVALID_CURSOR	ORA_01001	Nicht erlaubte CURSOR-Operation
INVALID_NUMBER	ORA-01722	Konvertierung eines Strings in eine Zahl schlug fehl
NO_DATA_FOUND	ORA-01403	SELECT INTO-Anweisung hat keine Daten zurückgeliefert
PROGRAM_ERROR	ORA-06501	Interner PL/SQL-Fehler (Hier ist „Holland in Not“, denn nun weiß noch nicht einmal die PL/SQL-Engine was falsch läuft.)
ROWTYPE_MISMATCH	ORA-06504	Spalte und Variable haben nicht den gleichen Datentyp
TOO_MANY_ROWS	ORA-01422	SELECT INTO-Anweisung liefert mehrere Zeilen
VALUE_ERROR	ORA-06502	Arithmetische Fehler, Konvertierungsfehler oder Zuweisungsfehler, z.B. Wert, der einer Variablen zugewiesen wird, ist zu lang
ZERO_DIVIDE	ORA-01476	Division durch Null
OTHERS		Steht für alle EXCEPTIONS, die nicht explizit im EXCEPTION-Handler aufgeführt wurden

Es gibt zwei Methoden, um EXCEPTIONS auszulösen:

- Eine EXCEPTION wird automatisch ausgelöst, wenn ein Oracle-Fehler auftritt, z.B. führt ORA-01403 die EXCEPTION „NO\_DATA\_FOUND“ aus.
- Sie können eine EXCEPTION auch explizit als Programmierer selbst mit der RAISE-Anweisung erzeugen. Auf diese Möglichkeit werden wir im Rahmen dieses Buchs nicht eingehen.

Es gibt zwei Methoden, um Fehlermeldungen im Programm anzuzeigen. Zwischen diesen beiden Methoden besteht ein kleiner, aber feiner Unterschied mit weit reichenden Folgen:

- DBMS\_OUTPUT.PUT\_LINE (Text)  
Dieser Befehl zeigt die Fehlermeldung formuliert als „Text“ auf dem Bildschirm an, sofern die Sessionvariable SERVEROUTPUT auf ON gesetzt ist. Wird dieser Befehl in den EXCEPTIONS verwendet, so wird das Programm zwar mit der Fehlermeldung, aber mit dem Status „successfully completed“ beendet.
- RAISE\_APPLICATION\_ERROR(negative\_nr, Text, { TRUE | FALSE } )  
Dieser Befehl zeigt auch eine Fehlermeldung an, die als Text vorgegeben wird. Zudem erzeugt er aber auch einen neuen Fehler und bricht das Programm fehler-

<sup>11</sup> Die vollständige Liste finden Sie in [Oracle PL/SQL 2005 Abschnitt 10, S. 10-4 ff.]

haft ab. Mit diesem Befehl erreicht man also, dass in den EXCEPTIONS ein Fehler behandelt wird und das Programm trotzdem fehlerhaft abbricht.

Die negative Zahl muss im Bereich von -20000 bis -20999 liegen. Sie spielt eine Rolle, wenn mehrere solcher Nachrichten zu unterscheiden sind und wenn Fehler bei geschachtelten Programmen zur späteren Bearbeitung nach oben weitergereicht werden.

Mit TRUE wird spezifiziert, dass die Fehlermeldung auf einem System-Fehler-Stack oben aufgelegt werden soll, und mit dem Default-Wert FALSE, dass die Meldung alle bisherigen Meldungen des Stack überschreibt.

### Beispiel

#### Division von Zahlen

Eine Funktion f\_division dividiert zwei Zahlen ungeprüft, die als Parameter übergeben werden. Das Ergebnis wird als Rückgabewert der Funktion zurückgegeben. Wird dabei ein Fehler „Division durch 0“ von der PL/SQL-Engine festgestellt, dann soll eine Fehlermeldung angezeigt werden, die Funktion soll aber nicht fehlerhaft terminieren (Rückgabe von NULL).

```
CREATE OR REPLACE FUNCTION F_Division (P_Divident IN NUMBER,
                                         P_Divisor IN NUMBER)
                                         RETURN NUMBER;
IS
BEGIN
    RETURN P_Divident/P_Divisor;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE ('Fehler: Division durch 0!');
        RETURN NULL;
END;
/
SHOW ERRORS;
-- korrekte Aufrufe
SELECT F_Division(24,6) FROM DUAL;
SELECT F_Division(20,3) FROM DUAL;
-- fehlerhafter Aufruf mit Divisor 0
SELECT F_Division(22,0) FROM DUAL;
```

### Beispiel

#### wieder eine Länder-Torstatistik mit SELECT INTO

 Dieses Beispiel greift das Beispiel „Eine Länder-Torstatistik mit SELECT INTO-Befehl“ aus Abschnitt 7.1.3 auf. Es wird die gleiche Aufgabe gelöst, nur diesmal wird der SELECT INTO, der den Übergabeparameter prüft, in einem anonymen Block mit eigener EXCEPTION geschachtelt.

```
CREATE OR REPLACE PROCEDURE Laender_Torstatistik_2 (P_Land IN VARCHAR2)
IS
    V_Min      NUMBER(3);
    V_Max      NUMBER(3);
    V_Avg      NUMBER(5,2);
```

```
BEGIN
    DECLARE
        V_Dummy      VARCHAR2(1);
    BEGIN
        SELECT 'x' INTO v_Dummy
        FROM Nation WHERE LOWER(Nationname) = LOWER(P_Land);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE_APPLICATION_ERROR(-20001,'Fehler!!!
                Das Land'||P_Land||' gibt es nicht oder hat nicht am Tunier teilgenommen!');
    END;
    SELECT MIN(Minute), MAX(Minute), AVG(Minute)
    INTO V_Min, V_Max, V_Avg
    FROM Tore
    WHERE Minute <999
    AND Spiel_Id IN (SELECT Spiel_Id
                      FROM Spiele
                      WHERE UPPER(Mannschaft_1) = UPPER(P_Land)
                      OR      UPPER(Mannschaft_1) = UPPER(P_Land));
    DBMS_OUTPUT.PUT_LINE('Für das Land'||P_Land||
                           ' gilt folgende Torstatistik');
    DBMS_OUTPUT.PUT_LINE('Maximale Torminute: '||V_Max);
    DBMS_OUTPUT.PUT_LINE('Minimale Torminute: '||V_Min);
    DBMS_OUTPUT.PUT_LINE('Durchschnittliche Torminute: '||V_Avg);
END;
/
SHOW ERRORS
EXEC Laender_Torstatistik('Paraguay');
-- Dieses Land hat nicht teilgenommen
EXEC Laender_Torstatistik('Luxemburg');
```

Um für verschiedene Anfragen in einem Programm individuelle Fehlermeldungen ausgeben zu können, ist die Kapselung in einen anonymen PL/SQL-Block mit eigener EXCEPTION Stand der Technik. Vergleicht man die EXCEPTION hier mit der ersten Lösung aus Abschnitt 7.1.3, so fällt auf, dass dort ein DBMS\_OUTPUT-Befehl verwendet wurde, während es hier ein RAISE\_APPLICATION\_ERROR ist. Würde man hier auch einen DBMS\_OUTPUT-Befehl nehmen, so hieße dies, dass bei einem Länderfehler der prüfende SELECT INTO abgebrochen und zur nächsten EXCEPTION gesprungt wird. Da dort der Fehler mit der NO\_DATA\_FOUND-EXCEPTION abgearbeitet wird, wird die Verarbeitung hinter dem END des anonymen Blocks ganz normal fortgesetzt, so dass nach der Fehlermeldung auch noch die vier anderen DBMS\_OUTPUT-Texte angezeigt würden, was sicherlich nicht wünschenswert ist. Ein Unterschied bleibt: Wenn ein falsches Land übergeben wurde, terminiert die erste Prozedur mit einer Meldung „successfully“ und die zweite bricht mit der Meldung fehlerhaft ab. Um diesen Unterschied aufzuheben, müsste in der ersten Prozedur statt des DBMS\_OUTPUT auch ein RAISE\_APPLICATION\_ERROR ausgeführt werden.

Werden Programme geschachtelt aufgerufen, dann besteht die Möglichkeit, Fehler aus unteren Programmebenen nach oben weiterzureichen und erst dort darauf zu reagieren. Mit diesem Thema können wir uns leider nicht weiter beschäftigen.

Damit z.B. bei Oracle-definierten und -erzeugten Fehlern keine Fehlerinformationen verloren gehen, ist es für selbst geschriebene Fehlerbehandlungen zentral, zwei Buildin-Funktionen zu kennen:

Tabelle 7.10

SQL-Fehlerfunktionen		
Fehlerfunktion	Beschreibung	Hinweise
SQLCODE	Nummer des Oracle-Fehlers	Negative Zahl mit Ausnahmen: 0 für kein Fehler +1 für benutzerdefinierten Fehler +100 für NO_DATA_FOUND
SQLERRM	Zugehöriger Fehlermeldungstext	512 Zeichen lang

Eine WHEN OTHERS-EXCEPTION ist unbedingt mit Vorsicht zu verwenden. Wird sie nicht richtig programmiert, können wichtige Fehlerinformationen verloren gehen. Will man dies vermeiden, so sollte man im Aktionsteil der EXCEPTION auf jeden Fall die beiden obigen Fehlerfunktionen verwenden, entweder als Anzeige auf dem Bildschirm oder als Teil der Fehlermeldung, die in einer Fehlerprotokolltabelle gespeichert wird.

### Beispiele

#### Anzeige der beiden Fehlermeldungen und fehlerfreies Terminieren des Programms

```
EXCEPTION
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE
    ('Fehler im Programm xyz: '||SQLCODE||' '||SQLERRM);
END;
```

#### Anzeige der beiden Fehlermeldungen und Abbruch des Programms:

```
EXCEPTION
  WHEN OTHERS THEN RAISE_APPLICATION_ERROR
    (-20001,'Fehler im Programm xyz: '||SQLCODE||' '||SQLERRM);
END;
```

## 7.2 Die Datenbankprogrammiersprache bei MySQL

Das noch sehr junge Alter der prozeduralen Erweiterung von MySQL (erst seit Version 5.0) wird am Sprach- und Funktionsumfang gegenüber dem von PL/SQL offensichtlich. Während PL/SQL schon auf eine lange Geschichte zurückblicken kann und immer neue Anforderungen vor allem auch von Benutzerseite an die Sprache und die Funktionsbibliotheken herangetragen wurden, orientiert sich MySQL sehr eng an den Vorgaben des Standards SQL2003 hinsichtlich der SQL-Invoked Routines<sup>12</sup> und ihrer Implementierung bei IBMs DB2. Die Kompatibilität mit dem DB2-Datenbanksystem

<sup>12</sup> Das Thema der „SQL-invoked routines“ – oder auch „PSM / persistent stored modules“ genannt – wird leider nur selten in Lehrbüchern behandelt. Zu finden ist es bei Melton und Simon in [Celko 2005], [Melton et al. 2002, S. 541-568] sowie Groff und Weinberg [Groff et al. 2002, S. 744-757].

wird ausdrücklich angestrebt, damit eine problemlose Portierung möglich ist<sup>13</sup>. Bei den gespeicherten Routinen („Stored Routines“) der Version 5 handelt es sich um eine allererste Implementierung, die noch unter recht vielen Restriktionen leidet. Wir weisen daher ausdrücklich darauf hin, dass die Praxiserfahrungen mit einem MySQL-Datenbanksystem Release 5.1.11. beta-log gesammelt wurden. Dieses Konzept wird jedoch weiterentwickelt, so dass die „Kinderkrankheiten“ sicherlich zukünftig verschwinden werden. Schaut man genauer hin, ist es doch überraschend, wie viel die gespeicherten Routinen von MySQL und Oracle bei der zentralen Funktionalität gemeinsam haben, weil sich beide an die Vorgaben von SQL2003 halten. Das MySQL-Kapitel ist nicht deshalb soviel dünner, weil wir – ausgenommen bei den Paketen – weniger Funktionalität erläutern, sondern weil wir an vielen Stellen auf die bereits bei PL/SQL gemachten Ausführungen verweisen können.

### 7.2.1 Datentypen und andere Grundlagen

MySQL unterstützt für seine gespeicherten Routinen die gleichen Datentypen wie in seiner Anfragesprache (vgl. Abschnitt 5.2.3). Außer den beiden Datentypen ENUM und SET gibt es keine komplexen Datentypen und benutzerdefinierte Datentypen sind auch in dieser prozeduralen Umgebung nicht implementiert. Dagegen sind Datumsdatentypen komfortabel und reichhaltig realisiert.

Da die Syntax bei den Blöcken keinen expliziten Deklarationsteil vorsieht, wird jede Deklarationsanweisung mit dem Schlüsselwort DECLARE eingeleitet. DECLARE-Anweisungen müssen zu Beginn eines Blocks stehen, vor allen anderen Anweisungen. Mit ihnen können lokale Variablen, „Conditions and Handler“ (vgl. Abschnitt 7.2.5) sowie CURSOR (vgl. Abschnitt 7.2.4) definiert werden. Es ist eine Reihenfolge zu beachten, zuerst alle Variablen und Conditions, dann die CURSOR und schließlich die Handler.

```
<Variablen-deklaration> ::=  
DECLARE Variablenname [, Variablenname]... <Datentyp> [ DEFAULT Wert ];
```

Der Datentyp kann ein beliebiger zulässiger MySQL-Datentyp sein, die wir bereits in Abschnitt 5.2.1 im SQL-Kontext erläutert haben. Der Defaultwert kann eine Konstante oder ein Ausdruck vom Typ sein. Der Gültigkeitsbereich einer solchen lokalen Variablen umfasst den Block, in dem sie deklariert ist, und alle geschachtelten Blöcke, außer denen, in denen eine Variable gleichen Namens deklariert wird.

### Beispiel

```
DECLARE V_Nachname CHAR(30) DEFAULT 'Meier';
DECLARE V_Telefon, V_Handy VARCHAR(20);
DECLARE V_Ergebnis DOUBLE DEFAULT 0;
DECLARE V_Datum DATE;
DECLARE V_Uhrzeit TIME;
DECLARE V_Bild BLOB;
```

<sup>13</sup> vgl. [MYSQL 2006, Kap. 19], [MYSQL Prozeduren 2005, S. 10]

```
<SET Anweisung> ::=  
  SET Variablenname = <Ausdruck> [, Variablenname = <Ausdruck>] ...;
```

Die Wertzuweisung erfolgt mittels der SET-Anweisung und Gleichheitszeichen „=“, wobei in einer Anweisung mehrere Zuweisungen durchgeführt werden können. Die Variablen können lokale definierte oder globale Systemvariablen sein, die wir hier aber nicht weiter vertiefen. Als Ausdruck sind Konstanten, Variablen oder Ausdrücke eines kompatiblen Datentyps zulässig.

Arithmetische, Vergleichs- und boolesche Operatoren, die implizite Typkonvertierung wie auch die Funktionen zur expliziten Konvertierung sind bereits in den Abschnitten 5.2.2 und 5.2.3 erläutert worden<sup>14</sup>. Für die Kurzschreibweise der Konkatenation von Zeichenketten, dem „||“, bleibt daran zu erinnern, dass sie nur im SQL-Modus „ANSI“ ausgeführt wird (vgl. SET SQL MODE-Anweisung im Abschnitt 5.3.3). Da die Routinen immer in dem Modus ausgeführt werden, in dem sie erzeugt wurden, ist es ausreichend, wenn solche Routinen im ANSI-Modus erzeugt werden.

Die Kurzform der SELECT-Anweisung ohne FROM-Klausel haben wir im Abschnitt 5.2.4 bereits vorgestellt, um SQL-Funktionen genau einmal auszuführen und den Rückgabewert auf dem Bildschirm anzeigen zu lassen. Dies funktioniert ebenso für selbst geschriebene Funktionen. Zudem kann dieses SELECT auch dazu verwendet werden, um in Prozeduren Meldungen auf dem Bildschirm auszugeben. Leider funktioniert dies nicht für Funktionen und Trigger.

## 7.2.2 MySQL-Routinen

Das MySQL-Konzept der gespeicherten Routinen kennt BEGIN END-Blöcke ohne Deklarationsteil, die als Ausführungsteil in Prozeduren, Funktionen und Triggern zu verwenden sind. Eine interaktive Verwendung als eine Art anonymer Block wie bei PL/SQL ist nicht möglich. Die Blöcke im Ausführungsteil lassen sich jedoch auch schachteln. Die Anweisungsliste ist optional, kann also auch leer sein (PL/SQL: Anweisung NULL;). Die einzelnen Anweisungen sind mit Semikolon voneinander zu trennen. Die Blöcke können mit einem Namen benannt werden.

```
<BEGIN END-Block> ::=  
  [ Blockname: ] BEGIN [ Anweisung [, Anweisung]... ] END [ Blockname ];
```

Bei MySQL werden die Prozeduren und Funktionen zusammenfassend als „gespeicherte Routinen“ bezeichnet. Im Weiteren werden wir daher auch nur zwischen Prozeduren und Funktionen differenzieren, wo dies notwendig ist, und ansonsten von Routinen sprechen.<sup>15</sup>

<sup>14</sup> Eine vollständige Liste aller Operatoren und ihrer Funktionalität finden Sie in [MySQL 2006, Kap. 12.1.1].

<sup>15</sup> Die Ausführungen dieses Abschnitts basieren im Wesentlichen auf der MySQL-Referenz [MySQL 2006] und dem Technical White Paper [MySQL Procedures 2005]

```
<CREATE PROCEDURE Anweisung> ::=  
  CREATE PROCEDURE Prozedurname ( [ <Prozedurparameterdefinition>  
    [, <Prozedurparameterdefinition>]... ] )  
    [ <Chrakateristika> [ <Chrakateristika>]... ]  
    <BEGIN END-Block>;  
  
<Prozedurparameterdefinition> ::=  
  [ IN | OUT | INOUT ] Parametername <Datentyp>  
  
<Chrakateristika> ::=  
  LANGUAGE SQL | (NOT) DETERMINISTIC | NO SQL | CONTAINS SQL  
  | READS SQL DATA | MODIFIES SQL DATA  
  
<CREATE FUNCTION Anweisung> ::=  
  CREATE FUNCTION Funktionsname  
    ( [ Parametername <Datentyp> [, Parametername <Datentyp>]... ] )  
    RETURNS type  
    [ <Chrakateristika> [ <Chrakateristika>]... ]  
    <BEGIN END-Block>;
```

Prozeduren und Funktionen unterscheiden sich hier nicht nur in der RETURNS-Klausel (Oracle: RETURN), sondern auch bei der Parameterdeklaration. Den Parametertyp IN | OUT | INOUT (Oracle: IN OUT) gibt es nur für Prozeduren. Hier ist die Reihenfolge erwähnenswert, erst der Typ IN | OUT | INOUT und dann Parametername mit Datentyp (Oracle: Parametername, Typ, Datentyp). Funktionen haben nur IN-Parameter, daher wird der Parametertyp nicht angegeben. Der einzige Rückgabewert wird über die RETURNS-Klausel zurückgegeben.

Routinennamen müssen eindeutig sein und maximal 64 Zeichen lang. Overloading ist nicht zulässig. Die Parameterliste ist immer obligatorisch. Ist sie leer, so werden sowohl bei der Erstellung als auch beim Aufruf nur die beiden Klammern hinter dem Routinenamen angegeben. Eine Besonderheit beim Aufruf von gespeicherten Routinen mit leerer Parameterliste ist, dass die beiden leeren Klammern ohne Leerzeichen auf den Prozedurnamen folgen müssen. Die Funktionalität der Parameter ist die gleiche wie bei PL/SQL, mit der Ausnahme, dass die übergebenen Werte der IN-Parameter innerhalb der Routine geändert werden dürfen. Diese Wertänderungen werden jedoch nicht außerhalb der Routine transparent.

Obwohl derzeit bei MySQL außer SQL keine andere Programmiersprache für die Routinen zulässig ist, muss die Option LANGUAGE SQL angegeben werden. Die übrigen Charakteristika müssen bei MySQL immer angegeben werden, wenn die Datenbank „binary logging“ durchführt, was für Recovery-Maßnahmen und Replikation der Fall ist<sup>16</sup>. Deterministisch ist eine Routine, wenn für die gleichen Eingabeparameter immer die gleichen Ergebnisse erzeugt werden. Alle anderen sind selbsterklärend: keine SQL-Anweisungen (NO SQL), enthält SQL-Anweisungen (CONTAINS SQL), liest bzw. schreibt Daten (READS SQL DATA bzw. MODIFIES SQL DATA). Auf die anderen Charakteristika verzichten wir hier, da sie nur zur Dokumentation und aus Kompatibilitätsgründen zu SQL aufgeführt und noch nicht vom Server genutzt werden.

Der Ausführungsteil (routine\_body) kann aus einer Anweisung bestehen oder aus einer Folge von Anweisungen, die dann aber zwischen BEGIN- und END-Anweisun-

<sup>16</sup> vgl. [MySQL 2006, Kap. 5.10.2.2, 6.2.2, 18.5] Bei Oracle findet man analoge Klauseln unter dem Stichwort „pragma restriction ini“ [Oracle PL/SQL 2005, S. 6-42, 8-23 ff.]

gen eingeschlossen werden müssen. Deklarationen werden mittels der DECLARE-Anweisung zu Beginn des Blocks durchgeführt (vgl. Abschnitt 7.2.1). Zulässig sind grundsätzlich alle Anweisungen der SQL-Anfrage-, Manipulations- und Datendefinitionssprache (vgl. Abschnitt 7.2.4) sowie die prozeduralen Anweisungen von MySQL.

Eine Funktionsausführung endet nicht einfach an der END-Anweisung, sondern benötigt zum Terminieren eine RETURN-Anweisung.

```
<RETURN Anweisung> ::=  
  RETURN [ Variablename | Wert ];
```

Um MySQL-Routinen programmieren zu können, muss zuvor ein Begrenzungszeichen (delimiter) definiert werden. Ein Problem ist, dass sowohl die interaktiv eingegebenen SQL-Anweisungen mit einem Semikolon abgeschlossen werden wie auch die Anweisungen im Ausführungsteil einer Routine. Um diese Problematik zu lösen, wird vor der ersten CREATE PROCEDURE/FUNCTION-Anweisung für SQL-Anweisungen ein neuer Delimiter definiert (Dieses Begrenzungszeichen gilt dann für den Rest der Sitzung für alle SQL-Anweisungen), als Ersatz für das sonst verwendete Semikolon. Mit dem gleichen Befehl kann anschließend das Semikolon wieder als Begrenzer eingesetzt werden. Der Backslash „\“ ist zu vermeiden, da er in MySQL das Escape-Zeichen ist.

### Beispiel

Als Begrenzungszeichen wird hier der Doppelslash „//“ definiert. Am Ende eines jeden interaktiv eingegebenen SQL-Befehls ist er nun anstelle des Semikolons zu verwenden.

```
DELIMITER //
```

Dieser Befehl setzt das Semikolon wieder in seine Funktion als SQL-Begrenzer ein.

```
DELIMITER ;
```

Aufgerufen wird eine Prozedur mit der CALL-Anweisung entweder in anderen Routinen oder interaktiv. Gespeicherte Funktionen werden aus anderen Anweisungen aufgerufen, wie die bereits bekannten PL/SQL-Funktionen auch (vgl. Abschnitt 6.1.2).

```
<CALL Anweisung> ::= CALL Procedurname;
```

### Beispiel

Schreiben Sie eine Prozedur „Mitteln“, an die zwei Zahlen x und y übergeben werden und die als Ergebnis  $(x + y) / 2$  liefert (vgl. auch PL/SQL, Abschnitt 7.1.2)!

```
DROP PROCEDURE Mitteln;  
DELIMITER //  
CREATE PROCEDURE Mitteln (IN x INT, IN y INT)  
BEGIN  
  DECLARE Ergebnis FLOAT;  
  SET Ergebnis = (x + y) / 2;  
  SELECT CONCAT('Ergebnis =', Ergebnis);
```

```
END; //  
CALL Mitteln (7,11) //  
DROP PROCEDURE Mitteln_Aufruf2//  
CREATE PROCEDURE Mitteln_Aufruf2 ()  
BEGIN  
  CALL Mitteln(22,26);  
END //  
CALL Mitteln_Aufruf2()//  
DROP PROCEDURE Mitteln_Aufruf//  
CREATE PROCEDURE Mitteln_Aufruf (IN x_auf INT, IN y_auf INT)  
BEGIN  
  CALL Mitteln(x_auf,y_auf);  
END //  
CALL Mitteln_Aufruf(5,7) //
```

### Beispiel

Schreiben Sie eine Funktion „doppeln“, die den übergebenen Wert verdoppelt zurückgibt! Da es sich um eine Funktion handelt, werden keine Parametertypen angegeben.

```
DELIMITER //  
DROP FUNCTION Func_Doppeln//  
CREATE FUNCTION Func_Doppeln (P_Zahl INT)  
RETURNS INT  
DETERMINISTIC  
BEGIN  
  RETURN P_Zahl*2;  
END //  
SHOW ERRORS//  
/* interaktiver Aufruf einer Funktion */  
SELECT Func_Doppeln (6)//  
/* Aufruf aus einer Prozedur */  
DROP PROCEDURE Func_Doppeln_Aufruf//  
CREATE PROCEDURE Func_Doppeln_Aufruf (IN Zahl INT)  
BEGIN  
  DECLARE V_Doppel INT;  
  SET V_Doppel = Func_Doppeln(Zahl);  
  SELECT V_Doppel;  
END//  
CALL Func_Doppeln_Aufruf(3) //  
DELIMITER ;
```

### Beispiel

Schreiben Sie eine Funktion „Gehaltssumme“, die das Gesamtgehalt über alle Angestellten ermittelt (vgl. PL/SQL, Abschnitt 7.1.2).

```
DELIMITER //  
CREATE FUNCTION Func_Gehaltssumme ()  
RETURNS INT  
READS SQL DATA  
BEGIN
```

```

DECLARE V_Summe INT;
SELECT SUM(Gehalt) INTO V_Summe FROM Angestellte;
RETURN V_Summe;
END //
SHOW ERRORS

/* interaktiver Aufruf einer Funktion */
SELECT Func_Gehaltssumme () //

/* Aufruf aus Aufruf einer Funktion */
CREATE PROCEDURE Func_Gehaltssumme_Aufruf ()
BEGIN
    DECLARE V_Summe INT;
    SET V_Summe = Func_Gehaltssumme();
    SELECT V_Summe;
END//
CALL Func_Gehaltssumme_Aufruf() //

```

### 7.2.2.1 Dynamisches SQL: Zur Laufzeit vorbereitete Anweisungen

Bei Oracle wird dynamisches SQL (NDS: EXECUTE IMMEDIATE) für die Ausführung von Datendefinitionsbefehlen und von zur Laufzeit zusammengesetzten Befehlen benötigt (vgl. Abschnitt 7.1.5). Bei MySQL wird dieser Begriff mit einer eingeschränkten Bedeutung verwendet. Dort können DDL-Anweisungen ohne spezielle Schnittstelle in den gespeicherten Routinen verwendet werden. Dynamisches SQL bezieht sich nur auf zur Laufzeit zusammengesetzte Anweisungen und heißt bei MySQL „Prepared Statements“. Bislang ist dieses Konzept jedoch nur in Prozeduren (Version 5.0.13) und nicht in Funktionen und Triggern verfügbar<sup>17</sup>. Die Erstellung und Ausführung von zur Laufzeit zusammengesetzten Anweisungen erfolgt in drei Schritten:

```

<PREPARE Anweisung> ::= 
    PREPARE Anweisungsname FROM <vorbereitete Anweisung>;

<EXECUTE Anweisung> ::= 
    EXECUTE Anweisungsname [USING @Variablenname [, @Variablenname] ... ];

<DEALLOCATE Anweisung> ::= 
    /DEALLOCATE | DROP| PREPARE Anweisungsname;

```

Mit der Anweisung PREPARE wird einer vorbereiteten Anweisung ein Name zugewiesen, unter dem die Anweisung später zugreifbar ist. Die <vorbereitete Anweisung> kann eine Zeichenkette sein oder eine Variable, die den Befehl enthält. Fragezeichen „?“ im Befehl stellen Übergabeparameter dar, die bei der Ausführung mit der EXECUTE-Anweisung mittels der USING-Klausel gefüllt werden können. Mit der Anweisung DEALLOCATE wird abschließend die Zuordnung zwischen Befehl und Namen wieder freigegeben. Bislang ist diese ganz neue Funktionalität erst für einige der SQL-Anweisungen verfügbar: CREATE TABLE, DELETE, DO, INSERT, REPLACE, SELECT, SET, UPDATE. An einer Ausdehnung auf weitere Befehle wird gearbeitet.

<sup>17</sup> vgl. [MySQL 2006, Kap. J.1]

### 7.2.2.2 Pakete

Das Oracle-Konzept der Pakete ist bei MySQL nicht bekannt, weder als benutzerprogrammierte Bibliotheken noch als Systembibliotheken, die vordefinierte Funktionalität bereitstellen. Trotzdem stellt MySQL eine große Vielzahl an Funktionalität in Form von Operatoren und Funktionen zur Verfügung. Eine vollständige Liste der sich ständig erweiternden Funktionalität enthält Kapitel 12 der SQL-Referenz [MySQL 2006]. Dort finden sich in erster Linie die SQL-Funktionen, aber auch Funktionalität, die Oracle als Paket zur Verfügung stellt.

Ein Beispiel sind die Verschlüsselungsfunktionen<sup>18</sup>. Mit DES\_ENCRYPT, DES\_DECRYPT, AES\_ENCRYPT, AES\_DECRYPT werden Verschlüsselungen mit dem 128-Bit Triple DES- (3DES) sowie dem 128-Bit (256-Bit) AES-Algorithmus durchgeführt (vgl. Abschnitt 7.1.2). ENCODE (string, pass\_string) und DECODE (string, pass\_string) verschlüsseln Zeichenketten auf der Basis von ebenfalls übergebenen Passwörtern.

### 7.2.3 Ablaufsteuerung und Kontrollstrukturen

In MySQL gibt es verschiedene Steuerungs- und Kontrollanweisungen:

- Sequenzen von Anweisungen, die durch Semikolon getrennt werden
- Bedingte Verzweigungen mit IF und CASE
- Verschiedene Schleifen: LOOP als Basisschleife ohne Bedingung, WHILE als kopfgesteuerte und REPEAT als rumpfgesteuerte Schleife
- Eine ITERATE-Anweisung, um einen erneuten Schleifendurchlauf anzustoßen
- Eine LEAVE-Anweisung zum Beenden von Schleifen

Auch hier beschränken wir uns auf einige grundlegenden Anweisungen, die es uns aber ermöglichen, beliebige Programme zu entwickeln.

Die Syntax und Funktionsweise der IF-Anweisung ist, bis auf ein „E“ bei ELSEIF, identisch zu der bereits bei PL/SQL vorgestellten (PL/SQL: ELSIF). Auch wenn der boolesche Datentyp BOOLEAN nur TRUE (<> „0“) und FALSE (= „0“) als Zustände zulässt (zweiwertige Logik; vgl. Abschnitt 7.2.1), so basieren doch sämtliche Vergleiche und booleschen Ausdrücke auf einer dreiwertigen Logik mit den Werten TRUE, FALSE, NULL.

Neben dieser SQL-konformen Variante des IF-Befehls gibt es noch die MySQL-IF-Funktion, die zu den „Control Flow Functions“ gehört.

```
<IF Anweisung> ::= IF ( <Ausdruck_1>, <Ausdruck_2>, <Ausdruck_3> )
```

Aufgerufen wird die IF-Funktion wie eine ganz normale Funktion, z.B. in einer SELECT- oder IF-Anweisung. Beim Aufruf wird für den ersten Ausdruck sein boolescher Wert ermittelt. Ist er TRUE (<> 0 und <> NULL), dann wird der zweite Ausdruck zurückgegeben, sonst der dritte.

<sup>18</sup> Eine vollständige Liste aller Verschlüsselungsfunktionen sowie ausführliche Erläuterungen dazu finden Sie in [MySQL 2006, Kap. 12.10.2].

**Beispiel**

```
SELECT IF(4711=4713,'ja','nein');
```

liefert das Ergebnis "nein".

Das Pendant zu der NVL-Funktion bei SQL (vgl. Abschnitt 5.2.3) ist bei MySQL die IFNULL-Funktion, die den ersten Ausdruck auf NULL prüft. Wenn er NULL ist, wird der zweite Ausdruck zurückgegeben, sonst der erste Ausdruck selbst.

```
<IFNULL-Funktion> ::= IFNULL ( <Ausdruck_1>, <Ausdruck_2> )
```

Eine solche „Control Flow Function“ gibt es dann auch noch für die SQL-konforme CASE-Anweisung sowie eine NULLIF-Funktion. Dieses Thema können wir hier aber leider nicht weiter vertiefen.

**Beispiel<sup>19</sup>**

```
CREATE PROCEDURE If_Test()
BEGIN
    DECLARE V_Anzahl INT;
    DECLARE V_Ausgabe CHAR(30);
    IF V_Anzahl = 0 THEN SET V_Ausgabe = 'A';
    ELSEIF V_Anzahl = 1 THEN SET V_Ausgabe = 'B';
    ELSEIF V_Anzahl = 2 THEN SET V_Ausgabe = 'C';
    ELSE SET V_Ausgabe = 'Weder A noch B noch C';
    END IF;
    SELECT V_Ausgabe;
END// 
CALL If_Test()//
```

Die Syntax der Basisschleife LOOP wie auch der WHILE-Schleife ist identisch zu der von PL/SQL (vgl. Abschnitt 7.1.4). Eine FOR-Schleife ist nicht realisiert und die REPEAT-Schleife, die ebenfalls identisch ist zu der in PL/SQL, wird hier auch nicht weiter behandelt.

Die Anweisung zum Verlassen einer Schleife heißt hier LEAVE und nicht EXIT WHEN wie unter Oracle. Anders als bei PL/SQL, wo die Bedingung zum Verlassen der Schleife in der EXIT-Anweisung formuliert wird, muss die LEAVE-Anweisung in einer bedingten Anweisung (IF, CASE) aufgerufen werden. Mit LEAVE können auch BEGIN END-Blöcke verlassen werden. Nur müssen die verlassenen Schleifen und Blöcke benannt sein.

```
<LEAVE Anweisung> ::= LEAVE Schleifename;
```

<sup>19</sup> vgl. PL/SQL, Abschnitt 7.1.4

Die ITERATE-Anweisung kann auch nur in einer benannten Schleife angewendet werden. Dort stößt sie explizit einen neuen Schleifendurchlauf an. Diese Anweisung ist gedacht für Fälle, in denen nicht bis zum Ende der Schleife weitergearbeitet werden soll, sondern bereits vorher entschieden wird, dass jetzt ein neuer Schleifendurchlauf ansteht.

```
<ITERATE Anweisung> ::= ITERATE Schleifename;
```

**Beispiel<sup>20</sup>**

```
BEGIN
    DECLARE V_Zaehler INT DEFAULT 0;
    Zaehlerschleife: LOOP
        SET V_Zaehler = V_Zaehler + 1;
        INSERT INTO V_Zaehler_Tabelle VALUES (V_Zaehler);
        IF V_Zaehler = 10 THEN LEAVE Zaehlerschleife;
        END IF;
    END LOOP Zaehlerschleife;
END;
```

**Beispiel**

In einer Prozedur soll für den übergebenen Wert die Fakultät berechnet werden (vgl. PL/SQL 7.1.4).

```
DROP PROCEDURE Fakultaet;
SET SQL_MODE='ANSI'// 
CREATE PROCEDURE Fakultaet (IN n INT)
BEGIN
    DECLARE Ergebnis BIGINT DEFAULT 1;
    DECLARE i INT DEFAULT 1;
    Fakult_while: WHILE i < n+1 DO
        SET Ergebnis = Ergebnis * i;
        SET i = i+1;
    END WHILE fakult_while;
    SELECT ('Die Fakultät von '||n||' ist: ' ||Ergebnis);
END; // 

SET SQL_MODE=''''
CALL Fakultaet (3)//
```

Um die Kurzschreibweise für die Konkatenation „||“ benutzen zu können, muss die Prozedur im ANSI-Modus erstellt werden. Das ist ausreichend, da gespeicherte Routinen immer in dem Modus ausgeführt werden, in dem sie erzeugt wurden.

<sup>20</sup> vgl. PL/SQL, Abschnitt 7.1.4

## 7.2.4 Datenbankzugriffe innerhalb von MySQL

In BEGIN END-Blöcken sind außer den SQL-Befehlen der Anfrage als SELECT INTO und den Manipulationsanweisungen INSERT, UPDATE und DELETE auch die Datendefinitionssprache mit den CREATE-, ALTER-, DROP-Anweisungen zulässig. In SQL2003 stellt dies ein optionales Feature dar und bei PL/SQL sind diese Befehle nur über „native dynamic SQL/NDS“ ausführbar (vgl. Abschnitt 7.1.5). Die einzigen unter MySQL unzulässigen DDL-Anweisungen sind:

CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE, CREATE FUNCTION, DROP FUNCTION, CREATE TRIGGER, DROP TRIGGER, LOCK und UNLOCK TABLES<sup>21</sup>

Nur diese seit Version 5 eingeführten DDL-Anweisungen sind (noch) nicht zulässig, alle anderen sind es sehr wohl. Dynamisches SQL, bei dem SQL-Anweisungen erst zur Laufzeit (vgl. Abschnitt 7.1.5) aus einzelnen Textbausteinen zusammengefügt werden, ist ganz neu seit Version 5.0.13 für Prozeduren zugelassen, aber weiterhin für Funktionen verboten.

Ferner sind nur für Funktionen ausgeschlossen: COMMIT, ROLLBACK sowie Anweisungen, die eine Datensatzmenge zurückliefern (außer SELECT INTO). Funktionen können nicht rekursiv aufgerufen werden und es gilt das gleiche Verbot, das aus dem Mutating Table-Problem bei Triggern resultiert (vgl. Abschnitt 7.3.5). In Funktionen darf die Tabelle, auf die bereits beim Aufruf der Funktion zugegriffen wurde, weder gelesen noch geändert werden. Um diese Einschränkung zu verstehen, muss man bedenken, dass Funktionen zum Beispiel sehr häufig innerhalb von SELECT-Anfragen oder in der SET-Klausel von UPDATE-Anweisung aufgerufen werden. Da ist es aus Gründen der Reihenfolgeunabhängigkeit (vgl. Abschnitt 7.3.6) nicht gestattet, während der Funktionsausführung auf diese gerade gelesene bzw. geänderte Tabelle zuzugreifen.

### Beispiel

 In einer Prozedur wird ein Index im Datenbankschema Rollo für die Tabelle „Spiele“ über die beiden Mannschaftsspalten definiert (vgl. MySQL, Abschnitt 7.2.4).

```
CREATE PROCEDURE WM_Index ()
BEGIN
    CREATE INDEX Spiele_Mannschaften_Idx
        ON Spiele(Mannschaft_1, Mannschaft_2);
END //
```

SHOW INDEX FROM Spiele//

Die DML-Anweisungen INSERT, UPDATE, DELETE werden unverändert in ihrer Syntax (vgl. Abschnitt 5.4) auch in den Routinen verwendet. Die bereits bei PL/SQL beschriebene SELECT INTO-Anweisung ist bis auf eine kleine Abweichung genauso bei MySQL realisiert. Da MySQL keine Record-Variablen kennt, können die selektierten Spaltenwerte nur lokalen Variablen zugewiesen werden. Die Funktionalität ist auch die gleiche, es kann nur genau ein Datensatz selektiert werden. Das CURSOR-Konzept, um eine ganze Datensatzmenge zu verarbeiten, ist ebenfalls völlig analog umgesetzt.

<sup>21</sup> Weitere hier nicht behandelte, aber ebenfalls unzulässige Anweisungen sind: LOAD DATA, LOAD TABLES, USE database, PREPARE, EXECUTE, DEALLOCATE PREPARE ...

### Beispiel

eine kleine Torstatistik mit SELECT INTO-Befehl

Es soll im Datenbankschema Rollo eine Prozedur geschrieben werden, die eine kleine Torstatistik betreibt und die maximale, minimale und durchschnittliche Torminute anzeigt, für alle Tore, bei denen die Zeit bekannt ist. Dabei steht die Torminute 999 wiederum für Tore, die durch Elfmeter erzielt wurden.

```
CREATE PROCEDURE Kleine_Torstatistik ()
BEGIN
    DECLARE V_Min      INT;
    DECLARE V_Max      INT;
    DECLARE V_Avg      FLOAT;
    SELECT MIN(Minute), MAX(Minute), AVG(Minute)
    INTO   V_Min, V_Max, V_Avg
    FROM   Tore
    WHERE  Minute < 999;
    SELECT( CONCAT('Maximale Torminute: ', V_Max));
    SELECT( CONCAT('Minimale Torminute: ', V_Min));
    SELECT( CONCAT('Durchschnittliche Torminute: ', V_Avg));
END; //
CALL Kleine_Torstatistik()//
```

Hier kann aus den gleichen Gründen, wie bei PL/SQL erläutert, auf die Behandlung von Fehlern verzichtet werden.

### Beispiel

eine Spieltagstatistik mit SELECT INTO-Befehl

Schreiben Sie eine Prozedur, der die Nummer eines Spieltags übergeben wird. Für diese übergebene Nummer wird zuerst getestet, ob es ein vorhandener Spieltag ist. Anschließend werden aus den Informationen der Ergebnisspalte die Summe der Tore und die Anzahl der unentschiedenen Spiele für diesen Spieltag ermittelt (vgl. PL/SQL, Abschnitt 7.1.5).

```
DELIMITER //
DROP PROCEDURE Spieltagstatistik//
```

```
CREATE PROCEDURE Spieltagstatistik (IN P_Tag  INT)
BEGIN
    DECLARE V_Torsumme      INT;
    DECLARE V_unentschieden INT;
    DECLARE V_Tag            INT;
    -- DECLARE EXIT HANDLER FOR 1172 BEGIN
    --     SELECT 'Zuviele Sätze gefunden!!!!' AS Fehler;
    -- END;
    DECLARE EXIT HANDLER FOR NOT FOUND BEGIN
        SELECT 'Kein Satz gefunden in der Spieltagstatistik!!!!' AS Fehler;
    END;
    -- Test, ob p_tag ein Spieltag ist
    SELECT  DISTINCT Spieltag
    INTO   V_Tag
    FROM   Spiele
    WHERE  Spieltag = P_Tag;
```

```

SELECT SUM(SUBSTR(Ergebnis,1,INSTR(Ergebnis,':')-1)
           + SUBSTR(Ergebnis,INSTR(Ergebnis,':')+1,LENGTH(Ergebnis)))
      INTO V_Torsumme
     FROM Spiele
    WHERE Spieltag = P_Tag;
SELECT CONCAT('Summe der Tore: ',V_Torsumme) AS Torsumme;

SELECT COUNT(*) INTO V_unentschieden
   FROM Spiele
  WHERE Spieltag = P_Tag
    AND SUBSTR(Ergebnis,1,INSTR(Ergebnis,':')-1) =
          SUBSTR(Ergebnis,INSTR(Ergebnis,':')+1,LENGTH(Ergebnis));
SELECT CONCAT
      ('Anzahl unentschiedener Spiele an dem Tag: ', V_unentschieden)
     AS Unentschiedene_Spiele;
END //

-- Test der Prozedur:
DELIMITER :
CALL Spieltagstatistik (1);
CALL Spieltagstatistik (2);

-- Diesen Spieltag gibt es nicht
CALL Spieltagstatistik (23);

```

In Vorgriff auf die im Abschnitt 7.2.5 ausführlich erläuterte Fehlerbehandlung wurden hier bereits zwei EXIT HANDLER verwendet. Der eine Fall wird ausgelöst, wenn ein Spieltag übergeben wird, der in der Tabelle nicht gespeichert ist, und somit kein Datensatz gefunden werden kann. Den anderen Fall, dass zu viele Sätze gefunden werden, brauchen wir hier nicht abzufangen, da die Anfragen nur ein Aggregatergebnis liefern und damit der Fehler nicht auftreten kann. Wir haben die Fehlerbehandlung für den zugehörigen Fehler 1172 auskommentiert. Wenn Sie die Option DISTINCT weglassen bei der Testanfrage, dann können Sie den Fehler 1172 erzeugen.

Das Konzept der CURSOR zur Lösung des „Impediment Mismatch“-Problems ist hier das gleiche wie bei PL/SQL (vgl. Abschnitt 7.1.5). Auch die MySQL-CURSOR sind „nur lesend“, die zwischengespeicherten Datensätze können also nicht verändert werden, und sie sind „nicht navigierend“, der Zeiger kann nur immer einen Satz vorrücken, nicht zurück und auch nicht mehrere Sätze überspringen (anders die RESULT SETS bei JDBC im Abschnitt 6.2.1).

Da sich sowohl Oracle als auch MySQL bei der CURSOR-Syntax an den SQL-Standard halten, gibt es bei den CURSOR-Befehlen keine Unterschiede. Die Anweisungen DECLARE, OPEN, FETCH, CLOSE sind identisch zu denen von PL/SQL. Um noch Platz und Zeit für andere interessante Features zu haben, verzichten wir hier auf eine wiederholte Darstellung der Syntax und verweisen wieder auf Abschnitt 7.1.5.

Lediglich die Art und Weise, wie festgestellt wird, ob alle Datensätze aus dem CURSOR ausgelesen wurden, differiert. Bei PL/SQL werden dafür die CURSOR-Variablen verwendet (vgl. Abschnitt 7.1.5). Bei MySQL erfolgt dies über den MySQL-Fehler mit der Kennung 1329 (MySQL Fehlererkennung) oder den SQLSTATE 02000 (Sqlstate-Wert)<sup>22</sup>. In Vorgriff auf die Fehlerbehandlung im nachfolgenden Abschnitt zeigen wir hier eines der PL/SQL-CURSOR-Beispiele in MySQL-Syntax.

<sup>22</sup> Die Fehlermeldung zum Error: 1329 SQLSTATE: 02000 (ER\_SP\_FETCH\_NO\_DATA) lautet:  
“Message: No data - zero rows fetched, selected, or processed”

### Beispiel 1

#### eines CURSOR

Geben Sie die ersten fünf Angestellten (in alphabetischer Reihenfolge) auf dem Bildschirm aus (vgl. PL/SQL, Abschnitt 7.1.5)!

```

DELIMITER //
SET SQL_MODE='ANSI'//
DROP PROCEDURE Ang_Fuenf// 
CREATE PROCEDURE Ang_Fuenf ()
BEGIN
    DECLARE V_Abt_Nr      INT;
    DECLARE V_Nachname   VARCHAR(20);
    DECLARE V_Zaehler    INT;
    DECLARE Ang_Cur CURSOR FOR SELECT Abt_Nr, Nachname
                               FROM Angestellte
                               ORDER BY Nachname;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET V_Zaehler = 5;
    SET V_Zaehler = 0;
    OPEN Ang_Cur;
    Ang_Schleife: LOOP
        FETCH Ang_Cur INTO V_Abt_Nr, V_Nachname;
        SET V_Zaehler = V_Zaehler + 1;
        IF V_Zaehler > 5 THEN LEAVE Ang_Schleife; END IF;
        SELECT CONCAT(V_Abt_Nr, ' ', V_Nachname) AS Abteilung_Mitarbeiter;
    END LOOP Ang_Schleife;
    CLOSE Ang_Cur;
END// 
SHOW ERRORS// 
CALL Ang_Fuenf()// 

```

Alternativ kann auch eine SELECT-Anweisung mit der LIMIT-Option<sup>23</sup> verwendet werden, um das gleiche Ergebnis zu erzeugen.

```
SELECT Abt_nr, Nachname FROM Angestellte LIMIT 5;
```

### Beispiel 2

#### eines CURSOR

Alle Angestellten werden abteilungsweise auf dem Bildschirm ausgegeben. Die Ausgabe wird abgeschlossen mit der Angabe der Gehaltssumme aller Angestellten sowie dem durchschnittlichen Gehalt aller Angestellten (vgl. PL/SQL, Abschnitt 7.1.5).

```

DROP PROCEDURE Angestellte_Ausgeben// 
CREATE PROCEDURE Angestellte_Ausgeben ()
BEGIN
    DECLARE V_Nachname  CHAR(20);
    DECLARE V_Gehalt    INT;
    DECLARE V_Abt_Nr    INT;
    DECLARE V_Summe    BIGINT;
    DECLARE V_Avg      BIGINT;

```

<sup>23</sup> vgl. Abschnitt 5.5.13

```

DECLARE V_Fertig CHAR(1) DEFAULT 'n';
DECLARE Ang_Cursor CURSOR FOR SELECT Abt_Nr, Nachname, Gehalt
    FROM Angestellte
    ORDER BY Abt_Nr;
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET V_Fertig = 'j';
OPEN Ang_Cursor;
WHILE V_Fertig = 'n' DO
    FETCH Ang_Cursor INTO V_Abt_Nr, V_Nachname, V_Gehalt;
    SELECT CONCAT
        (V_Abt_Nr, ' ', V_Nachname, ' ', V_Gehalt) AS
        Abteilung_Nachname_Gehalt;
END WHILE;
CLOSE Ang_Cursor;
SELECT SUM(Gehalt), AVG(Gehalt) INTO V_Summe, V_Avg
FROM Angestellte;
SELECT CONCAT
    (V_Summe, ' ', V_Avg) AS Gehaltssumme_Durchschnitt;
END//CALL Angestellte_Ausgeben()//

```

## 7.2.5 Fehlerbehandlung

MySQL verfolgt traditionell den Ansatz der Fehlervermeidung und der automatischen Korrektur, damit von einer Anweisung möglichst viel ausgeführt wird. Um ein Systemverhalten zu erreichen, das dem von SQL ähnelt – nämlich auf einen Fehler mit einem Abbruch zu reagieren –, muss der SQL-Modus entsprechend gesetzt werden. Ausführlicher werden die Modi im Abschnitt 5.1.3 zusammen mit den Integritätskonzepten vorgestellt. Hier werden wir meist den Modus TRADITIONAL verwenden, um die Routinen in einem Umfeld auszuführen, das ein zu SQL möglichst analoges Fehlerverhalten aufweist.

Die Fehlerbehandlung wird bei MySQL mittels „Conditions and Handlers“ realisiert. Tritt ein Fehler auf, der nicht behandelt wird, dann wird die Fehlerreaktion EXIT ausgeführt, was heißt, dass der ausgeführte BEGIN-END-Block beendet wird. Ein (Fehler-)Zustand wird benannt, der durch eine Fehlerbehandlung explizit bearbeitet werden soll.

```

<DECLARE CONDITION Anweisung> :=  

    DECLARE Fehlername CONDITION FOR  

        [ SQLSTATE [VALUE] <Sqlstate-Wert> | <Mysql Fehlerkennung> ];  

<DECLARE HANDLER Anweisung> :=  

    DECLARE [ CONTINUE | EXIT | UNDO | HANDLER  

        FOR <Fehlerwert> [, <Fehlerwert> ]... <BEGIN END-Block>];  

<Fehlerwert> ::= SQLSTATE [VALUE] <Sqlstate-Wert>  

    | Fehlername  

    | SQLWARNING  

    | NOT FOUND  

    | SQLEXCEPTION  

    | <Mysql Fehlerkennung>

```

Eine Fehlerbehandlung kann für mehrere Fehlerzustände programmiert werden, indem ihr *<Fehlerwert>* angegeben wird. Sobald einer der Fehler eintritt, wird sie ausgeführt, wobei es sich dabei um eine oder mehrere in einem BEGIN-END-Block (vgl. Abschnitt 7.2.2) zusammengefasste Anweisungen handeln kann. Wie nach der Fehlerreaktion fortgefahrene wird, hängt vom HANDLER-Typ ab. Ist CONTINUE spezifiziert, wird nach der Fehlerreaktion mit dem Programm fortgefahrene. Ist EXIT spezifiziert, wird die Ausführung für den BEGIN-END-Block beendet, in dem der HANDLER deklariert ist. Die UNDO-Option ist noch nicht implementiert.

Die Fehlerzustände können auf verschiedene Arten angesprochen werden, auch über vordefinierte HANDLER, wie wir dies schon bei PL/SQL gesehen haben:

- Eine SQLSTATE-Kennung *<Sqlstate-Wert>* für Server-Fehlermeldungen
- Ein mittels DECLARE CONDITION selbst vergebener Name *<Fehlername>* für eine SQLSTATE- oder MySQL-Fehlerkennung
- SQLWARNING als Abkürzung für SQLSTATE-Kennungen, die mit 01 beginnen
- NOT FOUND als Abkürzung für SQLSTATE-Kennungen, die mit 02 beginnen
- SQLEXCEPTION für alle übrigen SQL-Fehlermeldungen außer SQLWARNING und NOT FOUND
- Eine MySQL-Fehlerkennung *<Mysql Fehlerkennung>* für Server- und Client-Fehlermeldungen

Für die Server-Fehlermeldungen sind sowohl SQLSTATE- als auch MySQL-Fehlerkennungen vergeben, so dass sie über beide angesprochen werden können, wobei für viele MySQL-Fehler gilt, dass sie zu einer SQLSTATE-Kennung gruppiert sind. Arbeitet man also mit den MySQL-Fehlerkennungen, so hat man vielfach eine höhere Aussagekraft der Fehlermeldung.<sup>24</sup> Als problematisch hat sich in der Praxis insbesondere bei der Triggerprogrammierung erwiesen, dass noch keine Möglichkeit zur Erzeugung benutzerdefinierter Fehler besteht. Hier sind sicherlich neuere Versionen abzuwarten.

### Beispiel

#### Division von Zahlen

Eine Funktion F\_Division dividiert ungeprüft zwei Zahlen, die als Parameter übergeben werden. Das Ergebnis wird als Rückgabewert der Funktion zurückgegeben. Die Division durch den numerischen Wert 0 stellt in MySQL insofern kein Problem dar, als in den meisten SQL-Modi dafür kein Fehler erzeugt wird und als Ergebnis NULL angenommen und zurückgegeben wird. Lediglich im Modus TRADITIONAL führt dieser Fall zu einem Fehler. Da in Funktionen die Verwendung der SELECT-Anweisung zum Zwecke der Anzeige auf dem Bildschirm nicht zulässig ist, geben wir die Fehlermeldung einfach in einer Datei „Fehlermeldungen“ aus (vgl. PL/SQL, Abschnitt 7.1.5).

```

SET SQL_MODE = 'TRADITIONAL'//  

DROP FUNCTION F_Division//  

CREATE FUNCTION F_Division (Divident INT,  

                           Divisor INT)  

RETURNS FLOAT

```

<sup>24</sup> Eine vollständige Liste aller MySQL- und SQL-Fehlerkennungen finden Sie in [MySQL 2006, Kap. 12].