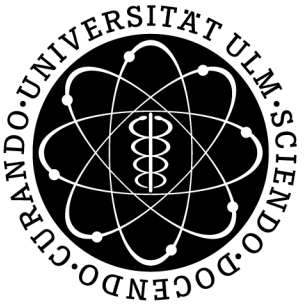
Diplomarbeit

Entwurf und Implementierung eines eNF2
Datenbank-Frontends für den Einsatz in der Lehre
HDBL Trainer

Dennis Benzinger
Dennis.Benzinger@gmx.net



Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme
Prof. Dr. Peter Dadam

18.05.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman: *Compilers*. Pearson Education, Inc., 2. Auflage, 2007, ISBN 0-321-49169-6.

[Blo08] Joshua Bloch: *Effective Java™*. Addison Wesley, 2. Auflage, 2008, ISBN 978-0-321-35668-0.

[GJSB05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha: *The Java Language Specification*. Addison Wesley, 2005, ISBN 978-0321246783. http://java.sun.com/docs/books/jls/index.html.

[Gooa] *Google Collections*. http://code.google.com/p/google-collections/.

[Goob] *Google Gson*. `http://code.google.com/p/google-gson/`.

[Lia99] Sheng Liang: *The Java™Native Interface*. Addison Wesley, 1999, ISBN 978-0201325775. `http://java.sun.com/docs/books/jni/`.

[PA86] Peter Pistor und F. Andersen: *Designing a Generalized NF2 Model With An SQL-Type Language Interface*. In: *Proceedings of the Twelfth International Conference on Very Large Data Bases*, 1986.

[Par07] Terence Parr: *The Definitive ANTLR Reference*. The Pragmatic Bookshelf, 2007, ISBN 978-09787392-5-6.

[PT85] Peter Pistor und Roland Traunmüller: *A Data Base Language for Sets, Lists, and Tables*. Technischer Bericht, Heidelberg Scientific Center – IBM Germany, 1985.

[RFC] *RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON)*. `http://www.rfc-editor.org/rfc/rfc4627.txt`.

[RQZ07] Chris Rupp, Stefan Queins und Barbara Zengler: *UML 2 Glasklar*. Carl Hanser Verlag, 3. Auflage, 2007, ISBN 978-3-446-4118-0.

[TS06] Can Türker und Gunter Saake: *Objektrelationale Datenbanken*. dpunkt.verlag GmbH, 2006, ISBN 3-89864-190-2.

[Tü03] Can Türker: *SQL:1999 & SQL:2003*. dpunkt.verlag GmbH, 2003, ISBN 3-89864-219-4.

[Ull09] Christian Ullenboom: *Java ist auch eine Insel*. Galileo Computing, 8. Auflage, 2009, ISBN 978-3-8362-1371-4. http://openbook.galileocomputing.de/javainsel8/.

Teil I

Ausarbeitung

Kapitel 1

Einführung

1.1 Aufgabenstellung

Die im Rahmen dieser Diplomarbeit bearbeitete Aufgabe war es, einen Prototypen eines HDBL Frontends für eine eNF2 Datenbank zu erstellen. Dieser Prototyp kann dann im Rahmen von Vorlesungen zur Durchführung von Übungen eingesetzt werden.

Damit der Prototyp auch von Studenten zu Hause eingesetzt werden kann, sollte darauf geachtet werden, dass bei Fehlern in ausgeführten HDBL Statements möglichst hilfreiche Fehlermeldungen ausgegeben werden. Weiterhin sollte das Programm wenige Abhängigkeiten haben und einfach zu installieren sein.

1.2 Grundbegriffe

HDBL Trainer
: Der HDBL Trainer ist die, im Rahmen dieser Diplomarbeit entwickelte, Software, mit der HDBL Anweisungen eingeben und gegen eine eNF2 Datenbasis ausgeführt werden können.

Basisdatentypen
: Basisdatentypen (atomare Typen) sind Datentypen, die durch das Datenbanksystem unstrukturiert gespeichert werden und für die das DBS keine Navigationsbefehle kennt. Beispiele sind Zahlen oder Wahrheitswerte.

Strukturierte Typen
: Sie haben einen, dem Datenbanksystem bekannten, inneren Aufbau und können deshalb in Teilen abgefragt und verändert werden. Liste oder Tupel sind Beispiele für strukturierte Typen.

Typkonstruktoren
: Mit ihnen können aus bestehenden Typen neue, strukturierte Typen er-

stellt werden. Zum Beispiel kann mit dem Typkonstruktor `LIST` und dem Basisdatentyp `INTEGER`, der Typ "Liste von Zahlen" (`LIST(INTEGER)`) erstellt werden.

Typkompositionsregeln
Typkompositionsregeln legen fest, wie sich Typkonstruktoren und Basisdatentypen kombinieren lassen (Kapitel 3.1 Grundkonzepte von Datenmodellen (S. 45) [TS06]).

1.3 eNF2 und HDBL

Klassische relationale Datenbanksysteme bieten mit Relationen, die in erster Normalform vorliegen müssen und mit SQL als Abfragesprache nur unzureichende Möglichkeiten, um hierarchische, komplex strukturierte Daten zu speichern. Neuere objektrelationale Erweiterungen des Datenmodells verbessern zwar die Möglichkeiten der Datenspeicherung, jedoch sind die objektrelationalen Erweiterungen von SQL zur Abfrage der Daten unvollständig und schlecht in die restliche Sprache integriert.

Hier setzt nun das eNF2 Datenmodell und die darauf aufsetzende Abfragesprache HDBL an. Abfragen von hierarchischen, komplex strukturierten Daten werden ebenso wie Abfragen von Daten in erster Normalform mit einer einheitlichen Syntax unterstützt.

eNF2 unterstützt die Typkonstruktoren Liste, (Multi-)Menge und Tupel. Basisdatentypen sind boolscher Wert, Zeichenkette und Ganzzahl. Die Typkompositionsregeln lassen eine beliebige Kombination der Typkonstruktoren untereinander zu. Nur die Blätter des Kompositionsbaumes müssen Basisdatentypen sein. Die Wurzel des Baumes darf durch jeden Typ und jeden Typkonstruktor gebildet werden.

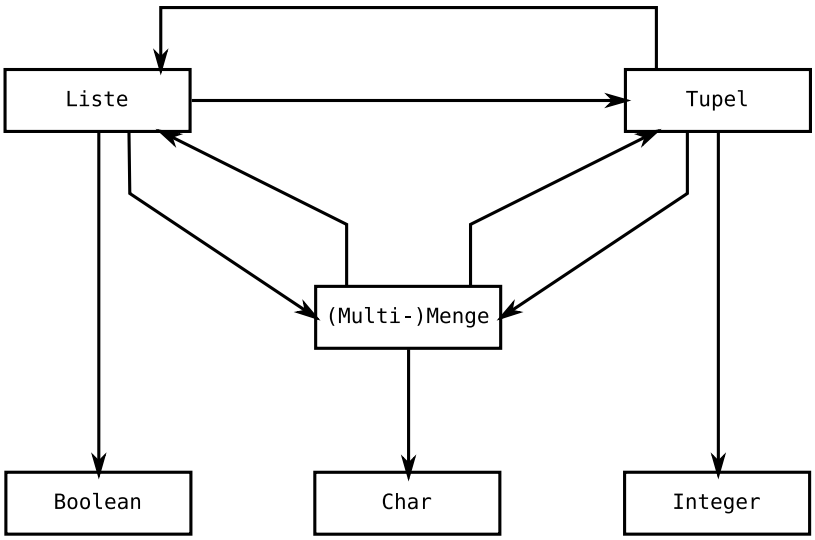


Abbildung 1.1: eNF2 Datentypen

Da eNF2 kein Objektmodell ist, haben die so gebildeten "Objekte" keine Indentität und keinen Zustand (Kapitel 3.3 – Objektmodelle (S. 66) [TS06]). eNF2 ist vielmehr ein wertbasiertes Datenmodell, wie zum Beispiel das Relationenmodell oder das Datenmodell von SQL-92. Diese kennen jedoch nur die Typkonstruktoren (Multi-)Menge und Tupel, und geben außerdem die Kombination der Typkonstruktoren fest vor: `(MULTI-)SET(TUPLE(Basisdatentyp))`

Die Heidelberg DataBase Language (HDBL, [PT85]) ermöglicht es, Daten des eNF2 Datenmodells komfortabel abfragen und bearbeiten zu können, ohne wie bei SQL hierarchische Daten im Anwendungsprogramm navigierend durchlaufen zu müssen.

Kapitel 2

Implementierung

2.1 Überblick

Der HDBL Trainer ist in Java programmiert und setzt die Java Laufzeitumgebung in Version 6 (Java™ SE Runtime Environment 6) oder höher voraus. Außer dem JRE™ 6 setzt der HDBL Trainer auch noch das Spracherkennungsframework ANTLR, die JSON Konvertierungsbibliothek Google Gson ([Goob]) und die Google Collections Erweiterung ([Gooa]) des Java Collections Frameworks ein.

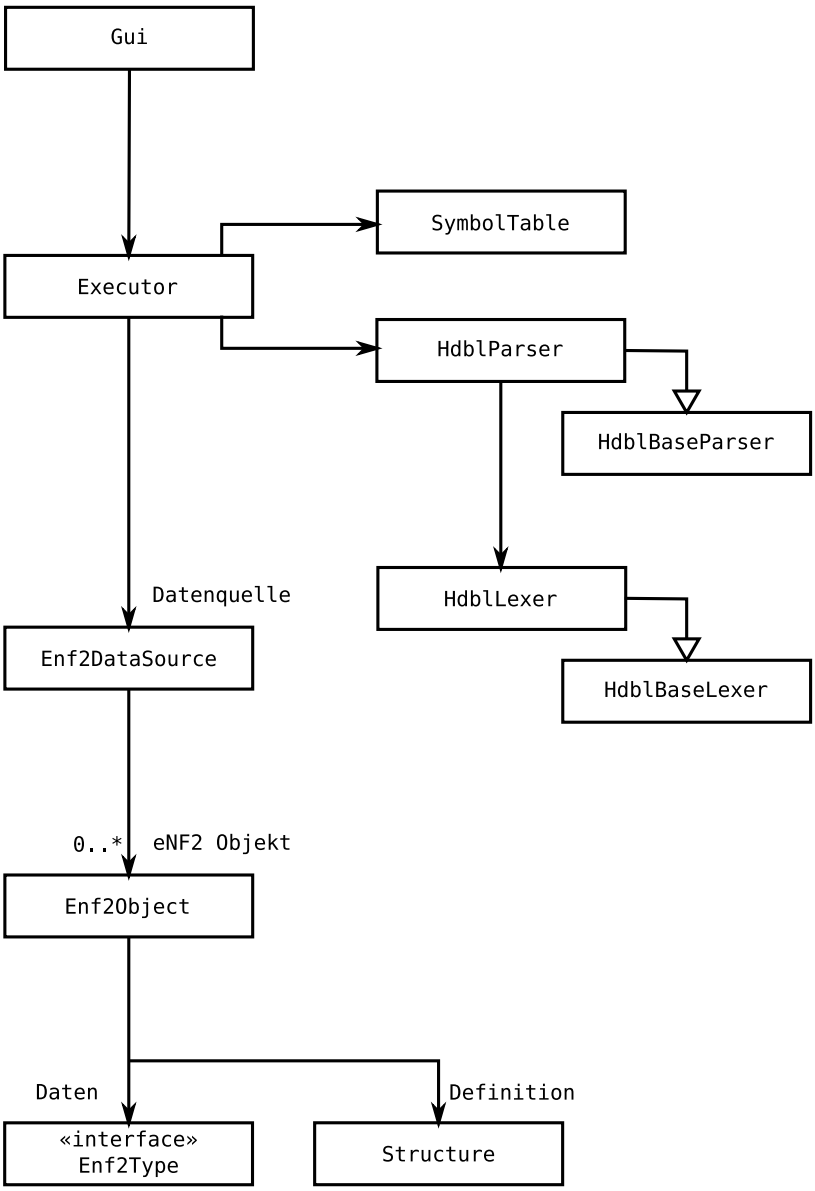


Abbildung 2.1: UML Übersicht

Der HDBL Trainer besteht im Wesentlichen aus drei Teilen: der Benutzerinteraktion, der Erkennung und Ausführung der Anweisungen und der Datenspeicherung.

2.2 Benutzerinteraktion

Die Benutzerinteraktion findet zur Zeit über eine einfache grafische Benutzeroberfläche statt. Aufgrund des modularen Aufbaus des HDBL Trainers ist es jedoch einfach möglich, diese Auszutauschen um zum Beispiel eine kommandozeilenbasierte Schnittstelle einzubauen. Auch ein Web-Interface wäre mit geringem Aufwand möglich.

Die GUI wurde mit Swing programmiert und besteht im Wesentlichen aus einem Textfeld für die Eingabe von HDBL Anweisungen und einem Textfeld für die Ausgabe der Daten einer `select` Anweisung oder einer Rückmeldung bei allen anderen Anweisungen. Die Rückgabedaten einer `select` Anweisung werden dabei in Textform angezeigt. Die Verschachtelung der Daten wird durch unterschiedliche Einrücktiefen der verschiedenen Hierarchieebenen verdeutlicht. Eine interaktive, graphische Darstellung ist derzeit nicht möglich. Diese könnte jedoch noch implementiert werden.

Um die Ausführung von wiederholt benötigten Anweisungen zu erleichtern, gibt es außerdem eine History, die zur Ausführung gebrachte Anweisungen aufzeichnet.

2.3 Datenspeicherung

2.3.1 Speicherung im Hauptspeicher

Um eNF2 Objekte zur Laufzeit zu verwalten, werden für diese Definition und Daten getrennt gespeichert. Dabei wird, neben den Klassen des Java Collection Frameworks, die Bibliothek Google Collections eingesetzt. Diese enthält weitere Klassen, die das Collection Framework ergänzen.

Definition

Der Aufbau eines eNF2 Objekts wird mit Hilfe von Objekten der `Structure` Klasse gespeichert. Diese Klasse besitzt die folgenden Felder:

`type: Class<? extends Enf2Type>`
: Dieses Feld legt den Typ fest, den das, durch dieses `Structure` Objekt beschriebene, eNF2 Teilobjekt haben muss. Das gespeicherte Objekt ist eine Instanz des `Class` Objekts der Klasse, die auch für die Speicherung der Daten dieses Teilobjekts benutzt wird.

  Durch die Benutzung des generischen Typs `Class<? extends Enf2Type>` ist sichergestellt, dass nur `Class` Objekte für Klassen gespeichert werden können, die das Interface `Enf2Type` implementieren.

Zum Beispiel wird für ein Teilobjekt, das eine Liste speichern soll, eine Instanz der Klasse `Class<Enf2List>` gespeichert.

`name: String`
: Für benannte Teilobjekte wird hier der Name gespeichert. Benannte Teilobjekte sind Tupelfelder oder das Teilobjekt eines eNF2 Objekts auf oberster Ebene. In diesem Fall ist `name` der Name des gesamten eNF2 Objekts. Unbenannte Teilobjekte haben ein `name` Feld, das `null` enthält.

`parent: Structure`
: Um einfacher in einem Baum von `Structure` Objekten navigieren zu können, speichern Unterobjekte eine Referenz auf ihr Elternobjekt. Dies ist zum Beispiel wichtig, um prüfen zu können, ob ein Teilobjekt ein Feld eines Tupels ist. Ohne eine `parent` Referenz müsste man dafür der Baum rekursiv durchlaufen und sich den jeweils aktuellen Weg von der Wurzel aus merken. Wenn dann das aktuelle Teilobjekt wiedergefunden wurde, könnte man aus dem Weg das Elternobjekt bestimmen. Um diesen Aufwand zu vermeiden, lohnt es sich, das `parent` Feld einzuführen.

`children: List<Structure>`
: Für `Structure` Objekte, die einen der strukturierten Typen `list`, `multiset` oder `tuple` repräsentieren, werden hier die Unterstrukturen gespeichert. Für Listen oder Multimengen enthält `children` genau eine Struktur. Für Tupel ist für jedes Feld eine Struktur vorhanden. Bei atomaren Typen ist `children` eine leere Liste.

`childrenNames: BiMap<String, Integer>`
: Um bei Tupeln schnell auf ein Feld eines bestimmten Namens zugreifen zu können, werden hier die Namen zusammen mit ihrer Position zwischengespeichert. Ohne diese Zwischenspeicherung müsste man alle Kindstrukturen durchlaufen und jeweils den Namen prüfen. Außerdem lässt sich so beim Anlegen einer neuen Kindstruktur leicht prüfen, ob ihr Name bereits verwendet wird.

  Durch die Verwendung einer `BiMap` ist sichergestellt, dass jede Position von genau einem Namen referenziert wird und nicht versehentlich zwei verschiedene Namen auf die gleiche Position verweisen.

Für ein eNF2 Objekt mit folgender Definition

```
create object schuh_lager {
    [
        modell: char,
        groessen: <
            integer
        >
    ]
}
```

wird zum Beispiel folgender Baum aufgebaut:

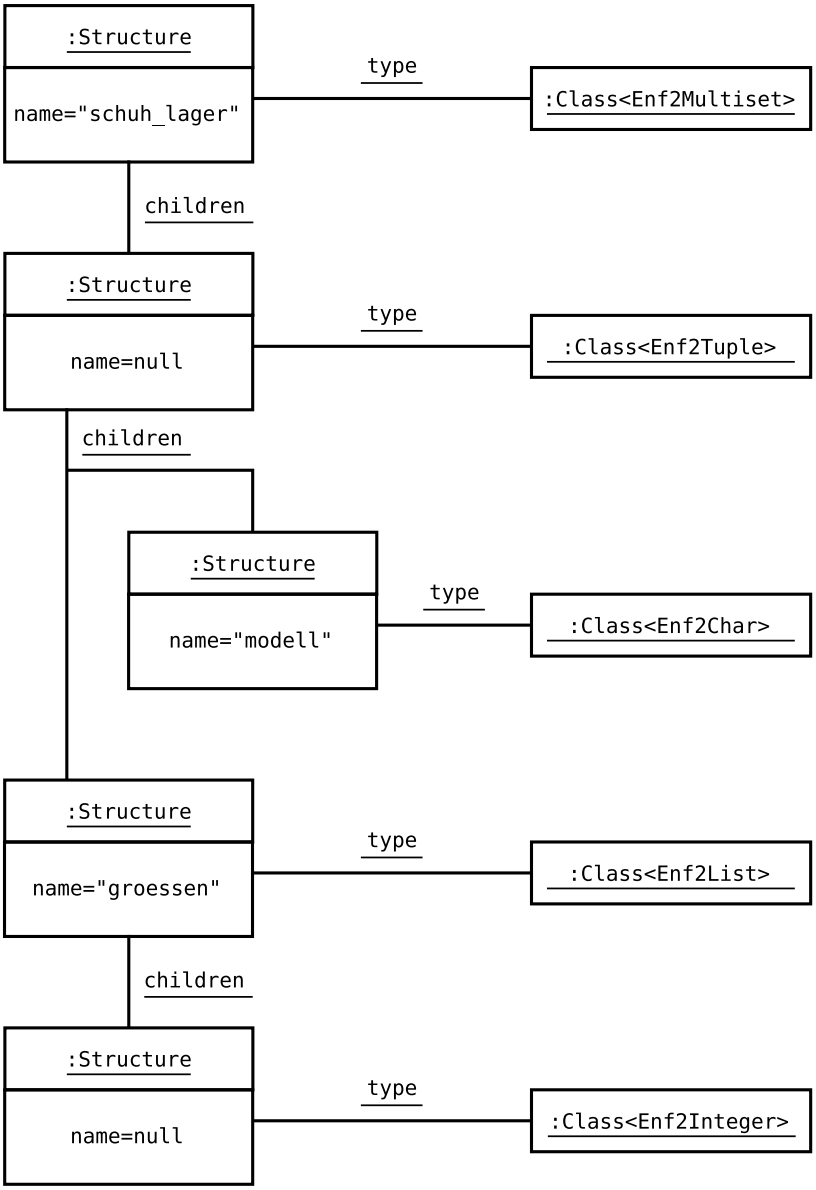


Abbildung 2.2: Beispiel `Structure`

Daten

Die Nutzdaten zu einem eNF2 Objekt werden mit Hilfe von Klassen des Java Collection Frameworks gespeichert. Zusätzlich wird die Google Collections Bi-

bliothek (siehe [Gooa]) verwendet, die das Collections Framework erweitern. Aus dieser Bibliothek werden hauptsächlich die Unterklassen von `ForwardingObject`, die Interfaces `BiMap` und `Multiset` benutzt. Diese werden im Weiteren noch näher beschrieben.

Um Parameter- und Rückgabetypen von Methoden und Typen von Feldern möglichst gut auf die Datentypen für eNF2 Daten einschränken zu können, werden die Collection Klassen nicht direkt eingesetzt. Vielmehr wurde eine spezielle Klassen- und Schnittstellen-Hierarchie implementiert, die genau die unterstützten eNF2 Datentypen abbildet.

Um eine Prüfung auf doppelte Datensätze und eine bei `select` Anweisungen eventuell notwendige Duplikateliminierung zu umgehen, unterstützt der HDBL Trainer keine Mengen wie in [PT85] beschrieben, sondern nur Multimengen, wie sie in [PA86] benutzt werden. Die anderen Datentypen, die der HDBL Trainer unterstützt, sind Liste, Tupel, boolscher Wert, Ganzzahl und Zeichenkette.

Die Vererbungs- und Implementierungsbeziehungen zwischen den Klassen sind in folgendem UML Diagramm der Übersicht halber zusammengestellt:

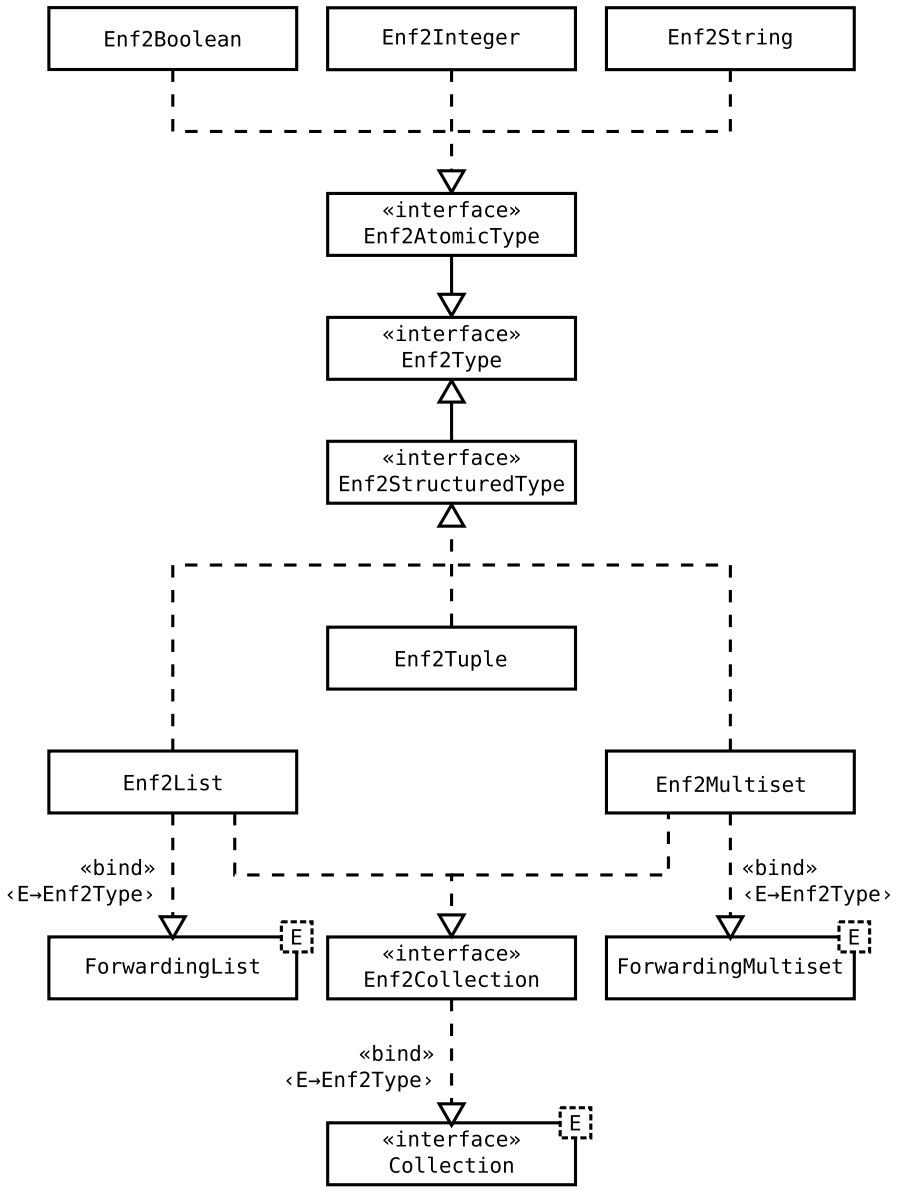


Abbildung 2.3: UML Datentypen

Um die Klassen, die für Listen und Multimengen zuständig sind, von den bereits vorhandenen Listen- und Multimengen-Klassen (zum Beispiel `java.util.List`) leichter unterscheiden zu können, wurden ihre Namen mit dem Präfix `Enf2` versehen. Um mit diesen Namen konsistent zu sein und damit man beim Programmieren nicht überlegen muss, ob das Präfix zu benutzen ist oder nicht, wurden

auch alle anderen Datenspeicherungsklassen mit dem Präfix ausgestattet.

Schnittstellen Die Basis der Klassen- und Schnittstellen-Hierarchie bildet die Schnittstelle `Enf2Type`. Sie dient dazu, Parameter, Rückgabewerte und Felder zu typisieren, wenn ein eNF2 Wert ganz allgemein beschrieben werden soll. Außerdem legt sie die folgenden beiden Methoden fest, die jede eNF2 Klasse implementieren muss:

`getDefinition(): Structure`
: Mit dieser Methode kann zu einem Datenobjekt die zugehörige Definition (siehe 2.3.1, S. 13) erfragt werden. Jedes Datenobjekt enthält eine Referenz auf ein `Structure` Objekt, das seine Struktur definiert. Diese Referenz verweist nicht auf die Definition des gesamten eNF2 Objekts, sondern genau auf das Objekt in der Strukturhierarchie, für das das Datenobjekt ein Element darstellt.

  In Fortführung des Beispiels zur `Structure` Klasse (Abb. 2.2, S. 15) hier ein Diagramm über die Datenobjekte zum eNF2 Objekt und ihre Verbindung zu den `Structure` Objekten:

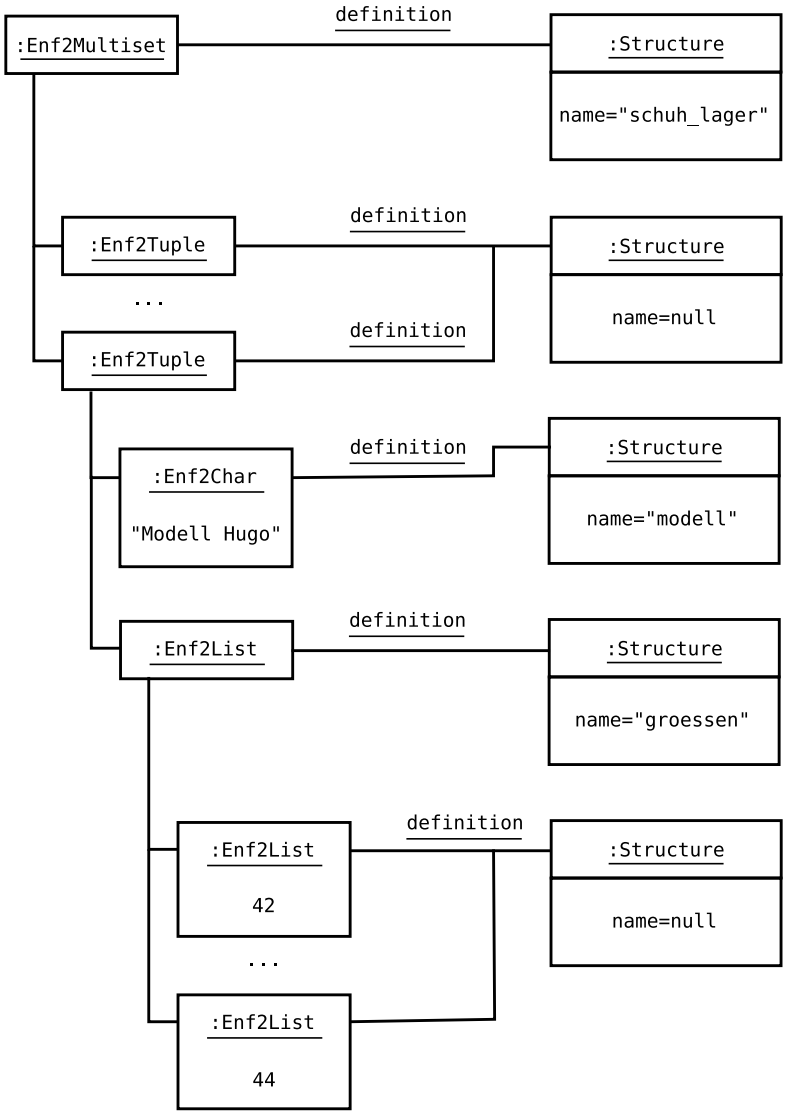


Abbildung 2.4: Beispiel Datentypen

Die `Structure` Objekte sind weiterhin wie am Anfang des Beispiels verbunden.

`setDefinition(Structure definition): void`
: Um Datenobjekte an eine Definition anzuhängen, kann diese Methode benutzt werden. Für Datentypen, die nicht atomar sind, werden auch die enthaltenen Datenobjekte angepasst. `setDefinition` wird zum Beispiel bei `extend` oder `insert` Anweisungen benutzt. Die beiden, bei diesen Anweisungen angegebenen Literale sind nicht vordefiniert, weshalb die aus

den Literalen erstellten Datenobjekte auch keine Definition referenzieren. Wenn die Datenobjekte schließlich in die Ziel-Liste oder -Multimenge eingefügt werden, müssen bei diesen die Referenz auf die Definition gesetzt werden.

Um atomare und strukturierte Datentypen einfacher unterscheiden zu können, wurden vom Basisinterface `Enf2Type` die Schnittstellen `Enf2AtomicType` und `Enf2StructuredType` abgeleitet. Diese enthalten keine Methoden und dienen somit nur als Marker-Interface.

Das Interface `Enf2Collection` ist ein weiteres Marker-Interface, das das generische Interface `Collection<E>` erweitert und dabei den Typparameter `E` an den Typ `Enf2Type` bindet. Dieses Interface wird benutzt, wenn eine Liste oder eine Multimenge verwendet wird, es jedoch nicht darauf ankommt, welcher dieser beiden Typen genau vorliegt. Zum Beispiel wird so in `for_each` Schleifen über Listen oder Multimengen iteriert.

Klassen Die atomaren eNF2 Typen `boolean`, `integer` und `char` werden von den Klassen `Enf2Boolean`, `Enf2Integer` und `Enf2Char` abgebildet. Diese enthalten jeweils ein Feld mit dem entsprechenden Java Typ: `Boolean`, `Integer` und `String`. Auf dieses Feld kann mit Getter- und Setter-Methoden zugegriffen werden.

Da es in Java für boolsche Werte, Ganzzahlen und Zeichenketten keine Interfaces, sondern nur Klassen gibt, konnten diese atomaren Datentypen nur über eine neue Klasse mit einer Referenz auf den eigentlichen Wert implementiert werden. Schöner wäre es gewesen, wenn für solche Typen Interfaces vorhanden wären. Dann hätten die atomaren Datentypen diese Interfaces implementieren können und man hätte sie wie normale Java Datentypen verwenden können.

Bei den Klassen für Listen und Multimengen ist dies jedoch der Fall. Für Listen ist im Java Collection Framework die Schnittstelle `List`, für Multimengen in Google Collections die Schnittstelle `Multiset` vorhanden. Da `Enf2List` und `Enf2Multiset` diese Schnittstellen implementieren, können sie überall dort eingesetzt werden, wo normale Listen und Multimengen erwartet werden. Dadurch ist die Verwendung dieser eNF2 Typen vereinfacht worden, da sie nicht bei jeder Verwendung in normale Java Listen oder Multimengen konvertiert werden müssen.

Um die für die Interfaces `List` und `Multiset` notwendigen Methoden nicht alle selbst implementieren zu müssen, basieren `Enf2List` und `Enf2Multiset` auf den Klassen `ForwardingList` und `ForwardingMultiset` aus der Google Collection Bibliothek. Diese Klassen stellen eine Implementierung der jeweiligen Schnittstelle dar, die alle Aufrufe von Methoden in diesem Interface an ein Objekt (genannt Delegate) einer Klasse, die das Interface implementiert weiterleiten. Die Subklasse muss dann nur noch das konkrete Objekt erzeugen, das die Aufrufe entgegennehmen soll. Im Falle von `Enf2List` ist dies eine Instanz von `ArrayList<Enf2Type>`. Methoden, deren Verhalten angepasst werden soll, können überschrieben werden und die überschreibende Methode kann

dann gegebenenfalls die Methode der Superklasse aufrufen. Durch den Einsatz der Forwarding Methoden ist es also einfach möglich, eigene Collection Klassen zu schreiben.

Um `Enf2List` und `Enf2Multiset` als eNF2 Datentypen zu kennzeichnen, implementieren auch sie die Schnittstelle `Enf2Type`. Ansonsten sind sie ganz normale Collection Klassen, die eine Menge von Elementen verwalten. Diese Elemente können auch doppelt vorkommen und sind im Falle von `Enf2List` geordnet, im Falle von `Enf2Multiset` ungeordnet.

Die letzte Klasse, die noch zur Datenspeicherung benutzt wird, ist für Tupel zuständig und trägt den Namen `Enf2Tuple`. Tupel bestehen aus einer, beim Erstellen des Objekts festgelegten Anzahl von Feldern, die eine optionale Bezeichnung tragen können. Im Gegensatz zu Listen und Multimengen können bei Tupeln keine neuen Elemente hinzugefügt werden, sobald das Tupel erstellt ist. Es ist nur noch möglich, die Werte einzelner Felder zu lesen und zu schreiben.

Eine Übersicht über die Beziehungen zwischen den Datenklassen und der `Structure` Klasse gibt folgende Abbildung:

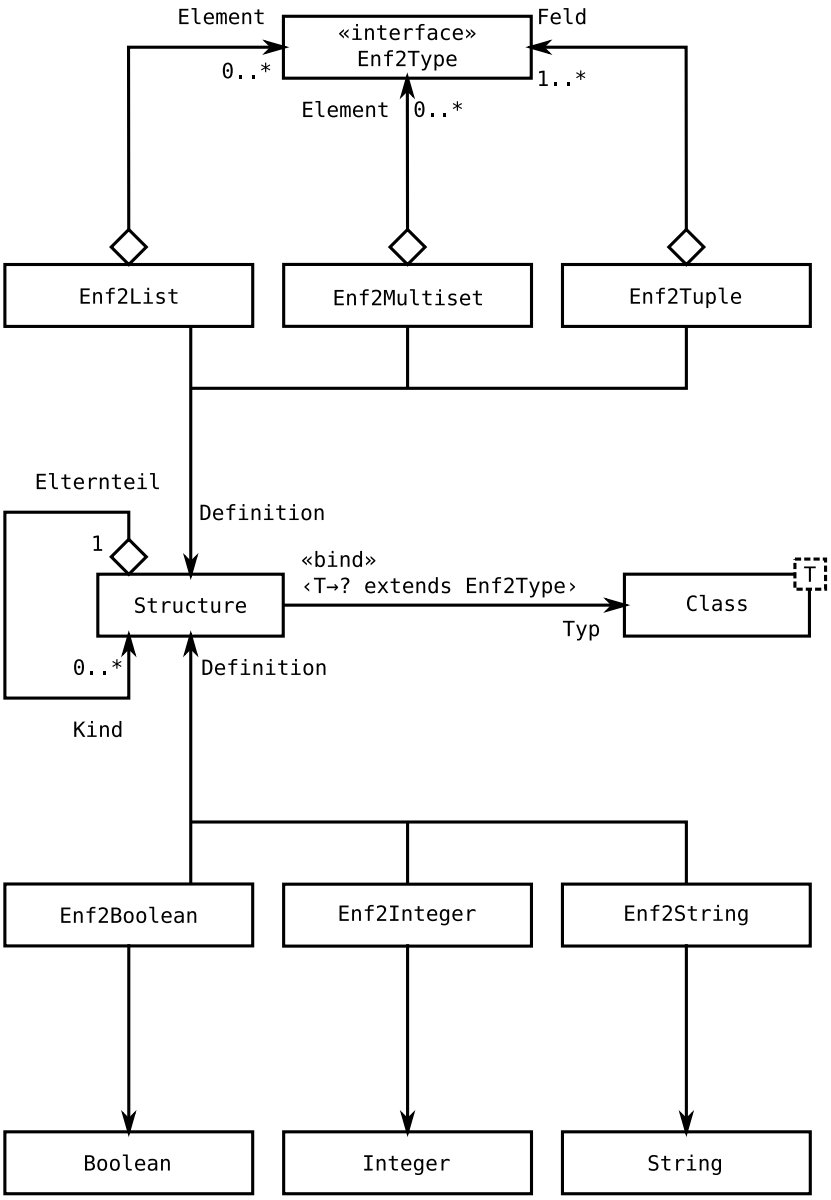


Abbildung 2.5: UML Datentypen-Assoziationen

2.3.2 Speicherung auf Datenträger

Für die persistente Speicherung auf Datenträgern wurde als erstes versucht, die eNF2 Datenobjekte in einer SQL Datenbank zu speichern. Dieses Vorgehen

erwies sich jedoch als zu umständlich, so dass schlussendlich die Speicherung in einer JSON Datei vorgezogen wurde.

SQL Datenbank

Damit der HDBL Trainer leicht zu installieren bleibt, konnte keine Datenbank mit einem separat zu installierenden Server (zum Beispiel Oracle Express Edition) benutzt werden. Daher wurde testweise die kleine, einbettbare Datenbank HSQLDB (`http://hsqldb.org/`) verwendet.

Um die Datenobjekte zu speichern, wurde für jedes benannte (Teil-)Objekt (Tupelfelder und toplevel Objekte) eine separate Tabelle angelegt. Diese bestand aus einem Datenteil und einem Strukturteil. Der Datenteil bestand bei Tupeln aus einer Spalte pro Feld und bei atomaren Objekten aus einer Spalte für den atomaren Wert. Listen und Multimengen wurden nicht in dem Datenteil abgelegt. Für sie kam der Strukturteil zum Einsatz. Für jede Liste oder Multimenge, in die ein Tupel oder ein atomarer Wert eingeschachtelt ist, wurde eine Listen- beziehungsweise eine Multimengen-Id-Spalte angelegt. Da Listen geordnet sind, wurde für sie außerdem eine `order` Spalte eingeführt. Damit die Zeilen in Tabellen für ein Tupelfeld das Tupel referenzieren können, haben Tupel-Tabellen auch noch eine Spalte für eine `tupel_id`.

Zum Beispiel werden aus dem eNF2 Objekt

```
create object robot {[
    rob_id: char,
    rob_descr: char,
    arms: {[
        arm_id: char,
        axes: <[
            dh_matrix: <<
                integer
            >>,
            ja_min: integer,
            ja_max: integer,
            mass: integer,
            accel: integer
        ]>
    ]},
    endeffectors: {[
        eff_id: char,
        function: char
    ]}
]}
```

folgende Tabellen

```
robot(tuple_id, rob_id)
robot_arms(tuple_id, parent_id, arm_id)
robot_arms_axes(tuple_id, order, parent_id,
```

```
    ja_min, ja_max, mass, accel)
robot_arms_axes_dh_matrix(parent_id,
    order_1, order_2, atomic)
robot_endeffectors(parent_id, eff_id, function)
```

Die Definition aller eNF2 Objekte wurde in einer Katalogtabelle mit folgendem Aufbau gespeichert:

Spalte	Spaltentyp	Beschreibung
`object_id`	`integer`	Id für das gesamte eNF2 Objekt
`structure_id`	`integer`	Id für ein Teilobjekt eines eNF2 Objekt
`name`	`string`	Name des Teilobjekts, null bei unbenannten Teilobjekten
`type`	`char`	Ein einzelnes Zeichen zur Kennzeichnung des eNF2-Typs
`parent_structure_id`	`integer`	Die `structure_id` des Elternobjekts

Tabelle 2.1: Katalogtabelle

Wie man an obigem Beispiel erkennen kann, ist schon für ein relativ gering verschachteltes eNF2 Objekt eine Vielzahl an Tabellen notwendig. Entsprechend aufwändig gestaltet sich die Abfrage und Bearbeitung der Daten. Eine weitere Schwierigkeit bestand darin, dass auch Unterobjekte zu Tabellen werden und diese nicht verschachtelt sind. Somit musste immer geprüft werden, ob der Name der Unterobjekte schon von einer anderen Tabelle benutzt wird. Dann hätte die Tabelle für das Unterobjekt automatisch umbenannt werden müssen und es hätte immer eine Namensumsetzung stattfinden müssen.

JSON

Wegen diesen Nachteilen einer SQL Speicherung wird nun stattdessen JSON benutzt. JSON (JavaScript Object Notation, [RFC]) ist ein schlankes, textbasiertes Datenaustauschformat, das auf der Syntax der Programmiersprache JavaScript basiert. Jede JSON Datenstruktur ist ein gültiger JavaScript Ausdruck. Es stehen die Notationen für Objekte, Arrays, Strings, Zahlen, boolsche Werte und den speziellen Null Wert zur Verfügung. Objekte sind in diesem Zusammenhang eine Menge von Name-Wert-Paaren.

Abbildung der Nutzdaten Diese Notationen stellen ziemlich direkt genau die Datentypen dar, die der HDBL Trainer unterstützt. Die eNF2 Datentypen wurden wie folgt auf die JSON Datentypen abgebildet:

`boolean`
: Boolsche Werte werden direkt als boolscher Wert in JSON gespeichert.

JSON kennt dafür die Literale `true` und `false`, die genau so auch in HDBL bekannt sind.

`char`
: Zeichenketten werden ebenfalls direkt im entsprechenden JSON Typ abgebildet. Nur schließt JSON Strings im Gegensatz zu HDBL in doppelten, anstatt in einfachen Anführungszeichen ein. Deshalb müssen in Zeichenketten vorkommende doppelte Anführungszeichen escaped werden. Auch einige andere Steuerzeichen wie Zeilenvorschub oder Backslash sind dabei zu beachten.

`integer`
: Auch Ganzzahlen lassen sich in JSON direkt abbilden. JSON würde sogar Kommazahlen unterstützen, welche der HDBL Trainer jedoch nicht benötigt.

`list`
: JSON kann auch Listen darstellen. Diese werden von JSON in eckigen Klammern eingeschlossen (zum Beispiel `[1, 2, 3]`) und nicht wie in HDBL in spitzen Klammern (`<1, 2, 3>`). Falls eine Liste leer sein sollte, wird sie in JSON auch als leere Liste dargestellt. Es wird nicht `null` für leere Liste benutzt.

`multiset`
: Multimengen werden bei der Speicherung ebenfalls als Arrays abgelegt. Um beim Laden einer Datenbank den Unterschied zwischen Listen und Multimengen feststellen zu können, wird ebenfalls gespeicherte Strukturinformation benutzt. Diese wird im folgenden Abschnitt noch näher beschrieben.

`tuple`
: Für Tupel werden die JSON Objekte benutzt. Diese Objekte besitzen Elemente, die einen JSON String als Name und einen beliebigen JSON Ausdruck als Wert haben. Zum Beispiel wird ein eNF2 Tupel `[name: 'Hans', alter: 5]` in JSON so dargestellt: `{"name": "Hans", "alter": 5}`.

Die Nutzdaten werden bei der Speicherung als verschachtelte JSON Ausdrücke gespeichert.

Strukturdaten Um die Definition von eNF2 Objekten speichern und laden zu können, muss diese ebenfalls in der JSON Datei abgelegt werden. Da Listen und Multimengen leer sein können und in diesem Fall keine untergeordneten Objekte existieren, genügt es nicht, nur die Daten zu speichern und aus ihnen beim Laden die Struktur wiederherzustellen. Dabei würde die Struktur der Unterobjekte verloren gehen. Deshalb speichert der HDBL Trainer die Struktur jedes eNF2 Objekts in JSON Objekten mit ab. Für jedes Objekt im Strukturbaum des eNF2 Objekt wird ein JSON Objekt mit folgenden Feldern erzeugt:

`type`
: Hier wird das `type` Feld des `Structure` Objekts abgebildet. Es wird als

String der eNF2 Name des Datentyps gespeichert. Also zum Beispiel für Multimengen `multiset`.

`name`
: Der Name des eNF2 (Teil-)Objekts wird direkt als JSON String gespeichert. Für namenlose Objekte wird `null` benutzt.

`children`
: Um die Sub-Strukturen eines Objekts zu speichern, wird dieses Feld benutzt. Es enthält ein Array von JSON Struktur-Objekten. Falls keine Sub-Strukturen vorhanden sind, ist das Array leer. `children` ist nie `null`.

Um auf die Nutz- und die Strukturdaten gemeinsam zugreifen zu können, werden die jeweiligen JSON Objekte in einem weiteren JSON Objekt mit den Feldern `structure` und `data` gekapselt.

Ein vollständiges Beispiel einer kleinen Datenbank als JSON Datei sieht dann wie folgt aus:

```
{
    "people": {
        "structure": {
            "type": "multiset",
            "name": "people",
            "children": [
                {
                    "type": "tuple",
                    "name": null,
                    "children": [
                        {
                            "type": "char",
                            "name": "given_name",
                            "children": []
                        },
                        {
                            "type": "char",
                            "name": "family_name",
                            "children": []
                        },
                        {
                            "type": "list",
                            "name": "cities",
                            "children": [
                                {
                                    "type": "char",
                                    "name": null,
                                    "children": []
                                }
                            ]
                        }
                    ]
```

```
                }
            ]
        },
        "data": [
            {
                "given_name": "Dennis",
                "family_name": "Benzinger",
                "cities": []
            },
            {
                "given_name": "Max",
                "family_name": "Mustermann",
                "cities": ["Berlin", "Hamburg", "Stuttgart"]
            }
        ]
    }
}
```

2.3.3 Die Klasse `Enf2DataSource`

Das Laden und Speichern dieser JSON Dateien übernimmt die Klasse `Enf2DataSource`. Sie verwendet dazu die Bibliothek Google Gson ([Goob]). Mit dieser Bibliothek lassen sich einfach JSON Dateien parsen und in einen Objektbaum verwandeln. Dieser wird dann durchlaufen und es werden für jedes eNF2 Objekt die Struktur und die Daten gelesen. Dabei wird zuerst die Strukturdefinition aus den JSON Objekten in `Structure` Objekte überführt. Anschließend werden die gespeicherten Nutzdaten gelesen, auf Konformität zur Struktur geprüft und in eNF2 Datenobjekte (zum Beispiel `Enf2List` oder `Enf2Boolean`) umgewandelt. Es werden immer alle in der Datei gespeicherten Objekte in den Hauptspeicher eingelesen. Da die Datenmengen klein sein werden, stellt dies jedoch kein Problem dar.

Beim Speichern werden aus den eNF2 Objekten wieder JSON Objekte erstellt, wie sie in 2.3.2 Nutzdaten und 2.3.2 Strukturdaten beschrieben sind. Dann werden die JSON Objekte im JSON Textformat in eine Datei geschrieben. Da die zu speichernden Datenmengen gering sind, werden nicht nur die Veränderungen (zum Beispiel neue und gelöschte Daten) geschrieben, sondern immer alle eNF2 Objekte gespeichert.

2.4 Erkennung und Ausführung der Anweisungen

Um die vom Benutzer eingegebenen Anweisungen ausführen zu können, müssen diese von einem Lexer und einem Parser erkannt und gleichzeitig darauf geprüft werden, ob sie in der Teilmenge von HDBL enthalten sind, die der HDBL Trai-

ner unterstützt. Damit der dafür verwendete Lexer und Parser nicht vollständig von Hand geschrieben werden muss, ist der Einsatz eines Spracherkennungsframeworks sinnvoll.

2.4.1 Erkennung der Anweisungen

Die Erkennung der Anweisungen sollte durch den Einsatz eines Spracherkennungsframeworks vereinfacht werden. Ein solches Spracherkennungsframework besteht aus einem Programm und eventuell benötigten Bibliotheken, die aus einer Grammatik, die die zu erkennende Sprache beschreibt, einen Lexer und einen Parser für diese Sprache erstellt. Dieser wird dann vom eigentlichen Anwendungsprogramm benutzt, um Eingaben, die der Sprache entsprechen, zu erkennen.

Da der HDBL Trainer in Java programmiert ist, musste ein Spracherkennungsframework ausgewählt werden, das mit dieser Programmiersprache einsetzbar ist. Dadurch schied zum Beispiel schon der bekannte Lexergenerator Lex und der Parsergenerator Yacc aus, da diese nur C-Code (oder in neueren Versionen auch C++ Code) erzeugen können. Dieser hätte zwar über das Java Native Interface ([Lia99]) angebunden werden können, allerdings hätte er dann für jede Plattform, auf der der HDBL Trainer laufen soll, kompiliert werden müssen. Dies steht im Widerspruch zur einfachen Installierbarkeit, die für den HDBL Trainer gefordert war. Außerdem wäre dadurch die Entwicklung unnötig verkompliziert und durch die Verwendung einer Programmiersprache ohne automatische Speicherverwaltung fehleranfälliger geworden.

Es gibt aber auch mehrere Lexer- und Parsergeneratoren, die Java als Ausgabesprache unterstützen. Unter anderem sind dies Cup / JFlex (`http://www2.cs.tum.edu/projects/cup/`, `http://jflex.de/`), JavaCC (`https://javacc.dev.java.net/`) und ANTLR.

Von diesen Alternativen ist ANTLR am besten dokumentiert und wird am kontinuierlichsten weiterentwickelt. Außerdem ist dafür ein Buch mit umfangreicher Erläuterung der Entwicklung mit ANTLR vorhanden. Zur weiteren Vertiefung wird dort auch die Theorie hinter den generierten Lexern und Parsern erläutert. Auch gibt es zur Unterstützung bei Problemen eine lebhafte Mailingliste (`http://www.antlr.org/support.html`). Aus diesen Gründen wurde schlussendlich ANTLR benutzt.

ANTLR

ANTLR (ANother Tool for Language Recognition, `http://www.antlr.org`) ist ein Werkzeug, mit dem man Lexer und Parser, für eine, durch eine Grammatik definierte Sprache, erzeugen kann. ANTLR wurde von Associate Professor Terence Parr (jetzt University of San Francisco) erfunden. ANTLR selbst ist zwar in Java geschrieben, kann aber, im Gegensatz zu vielen anderen Werkzeugen zur Spracherkennung, Lexer und Parser für viele verschiedene Ausgabesprachen

erzeugen. Die wichtigsten Sprachen die ANTLR unterstützt sind Python, Java, C, C#, JavaScript und ActionScript. Eine Liste der unterstützten Sprachen findet sich im Web unter `http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets`. Auch ist die Unterstützung einer weiteren Sprache relativ einfach möglich. Eine Anleitung dazu findet sich unter `http://www.antlr.org/wiki/display/ANTLR3/How+to+build+an+ANTLR+code+generation+target`.

Grammatiken Die Angabe der Grammatiken erfolgt in der erweiterten BNF. Dadurch werden optionale und sich wiederholende Elemente direkt unterstützt und müssen nicht über Alternativen oder Rekursion abgebildet werden. Das folgende Beispiel zeigt diesen Vorteil anhand eines Ausschnitts einer Grammatik einer SQL-ähnlichen Definition von Spalten:

```
col_list: col (COMMA col)*;
col: id COLON datatype col_spec?;
col_spec: (NOT_NULL | PRIMARY_KEY)+;
```

Ohne EBNF Unterstützung ergäbe sich folgende, deutlich unübersichtlichere Grammatik:

```
col_list: col | col COMMA col_list;
col: id COLON datatype | id COLON datatype col_spec;
col_spec: NOT_NULL | PRIMARY_KEY | col_spec col_spec;
```

Eine weitere Erleichterung bei der Erstellung von Grammatiken bietet ANTLR mit der Unterstützung von Subregeln. Dies sind in Klammern eingeschlossene Elemente auf der rechten Seite einer Regel. Sie können benutzt werden, wenn diese Elemente nur in einer Regel benötigt werden und deshalb die Grammatik durch die Definition einer separaten Regel unnötig aufgebläht werden würde.

2.4.2 Ausführung der erkannten Anweisungen

Abstrakter Syntaxbaum

Um mit ANTLR Aktionen für einen erkannten Text ausführen zu können, gibt es zwei grundsätzlich verschiedene Möglichkeiten. Entweder man gibt in der Grammatik direkt den Code an, der für bestimmte Regeln ausgeführt werden soll, oder man lässt ANTLR einen abstrakten Syntaxbaum (Abstract Syntax Tree, AST) erzeugen, der dann die weiter Verarbeitung steuert.

Die erste Methode mit dem Code in den Regeln eignet sich vorallem für kleinere, lokale Aktionen. Denn wenn viel Code auszuführen ist, wird die Grammatik durch die Menge des Codes und durch seine Durchmischung mit den eigenlichen Regeln sehr unübersichtlich. Auch Aktionen, die andere Teile des Quelltexts beachten müssen, sind so schwierig oder gar nicht möglich. Falls vorhergehender Code gebraucht wird, kann man diesen noch von Hand speichern, falls jedoch nachfolgender Code benötigt wird, ist dies nicht möglich. Bei HDBL tritt dieser

Fall zum Beispiel bei `select` Anweisungen auf. Wenn man dort eine Aktion für das `select` Element ausführen wollte, so müsste man auch Zugriff auf die erst später erkannte `from` Klausel haben.

Deshalb scheidet diese Möglichkeit für den HDBL Trainer aus. Stattdessen lässt er sich von ANTLR einen abstrakten Syntaxbaum generieren und arbeitet diesen dann selbstständig ab. Solch ein Syntaxbaum besteht aus Knoten, die Teile des Eingabetextes abbilden, die für die weitere Verarbeitung notwendig sind und diese werden in eine dafür günstige Reihenfolge gebracht. Bestandteile des Eingabetextes, die nur für die Erkennung der Struktur notwendig sind, werden dabei weggelassen. Zum Beispiel werden die Schlüsselwörter `do` und `end` bei einer `for_each` Anweisung weggelassen, da sie nur dazu dienen, die Unteranweisung zu begrenzen. Stattdessen wird die hierarchische Struktur der Anweisung direkt über die Struktur des Baums abgebildet. Dies sieht dann wie folgt aus:

Aus der Anweisung `for_each e in l where e <= 4 do e := 5 end` wird folgender Baum:

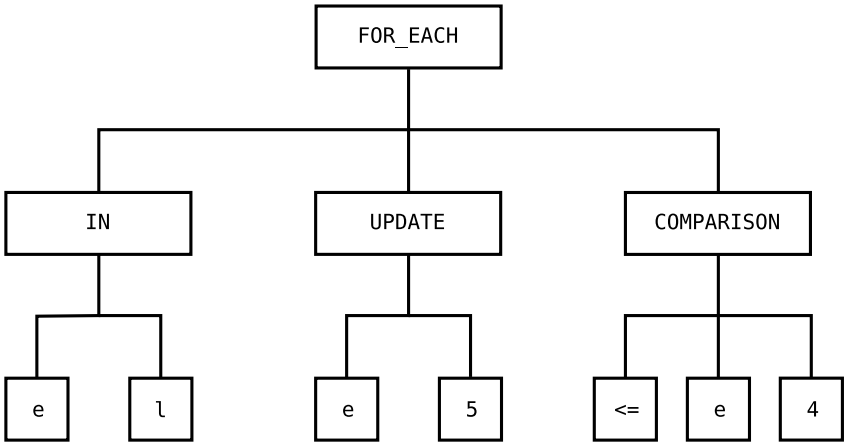


Abbildung 2.6: Beispiel AST

Außer den fehlenden Schlüsselwörtern `where`, `do` und `end` erkennt man hier auch, das die `where` Klausel an den Schluss gestellt wurde. Dies wird gemacht, da die Bedingung nicht angegeben werden muss und sie in diesem Fall einfach im AST weggelassen werden kann ohne die Positionen der anderen Kinder des `FOR_EACH` Knoten zu verändern.

Executor

Der Executor ist ein Objekt der Klasse `Executor`, die Methoden zur Verarbeitung der Anweisungen im AST enthält. Die Einstiegsmethode im Executor ist `execute`. Sie erhält die eingegebene Anweisung als String, kümmert sich um die gesamte Ausführung und gibt schlussendlich das Ergebnis zurück. Für Anweisungen, die kein Ergebnis haben, wie zum Beispiel `insert`, wird eine Rückmeldung über die Durchführung der Anweisung gegeben.

Als erstes übergibt `execute` die Anweisung an den Parser. Dieser versucht sie zu erkennen und den AST dafür aufzubauen. Wenn der Parser den abstrakten Syntaxbaum fertiggestellt hat und dabei kein Fehler aufgetreten ist, dann fragt `execute` den Typ der Anweisung ab und leitet den AST an die für den jeweiligen Anweisungstyp zuständige Methode weiter. Es gibt die Methoden `createObject`, `describeObject`, `select` und `dmlStatement`. Die ersten drei sind genau für die jeweilige Anweisung zuständig. Die letzte Methode ist für die DML Anweisungen `delete`, `extend`, `insert`, `:=` (Update) und `for_each` zuständig. `for_each` wird auch durch `dmlStatement` behandelt, da damit weitere DML Anweisungen ausgeführt werden.

Diese Methoden werden nun im Detail beschrieben:

`createObject` Diese Methode verarbeitet den abstrakten Syntaxbaum einer `create object` Anweisung. Die im AST beschriebene Struktur des zu erstellenden Objekts wird dabei in einen Baum von `Structure` Objekten (siehe 2.3.1, S. 13) abgebildet. Dieser wird dann an die benutzte Datenquelle übergeben, die dafür die Basisdatenobjekte anlegt und diese zusammen mit der Strukturdefinition ablegt. Die Basisdatenobjekte (BDO) für die eNF2 Datentypen sind folgende:

eNF2 Typ	Klasse BDO	Wert BDO	Beschreibung
`boolean`	`Enf2Boolean`	`false`	
`char`	`Enf2Char`	`''`	leerer String
`integer`	`Enf2Integer`	`0`	die Zahl Null
`list`	`Enf2List`	`<>`	leere Liste
`multiset`	`Enf2Multiset`	`{}`	leere Multimenge
`tuple`	`Enf2Tuple`		ein Tupel, bei dem für jedes Feld ein BDO angelegt wird.

Tabelle 2.2: Basisdatenobjekte (BDO)

`describeObject` Hier wird über die Datenquelle das eNF2 Objekt zu dem im AST angegebenen Namen gesucht und dafür die Strukturdefinition gelesen. Diese wird dann in einen String umgewandelt, der das gleiche Format hat wie die Strukturbeschreibung in einer `create object` Anweisung.

`select` Da `select` Anweisungen verschachtelt sein können und somit eine rekursive Ausführung notwendig ist, wurde dafür die spezielle Klasse `SelectExecutor` entwickelt. Die `select` Methode erstellt bei ihrem Aufruf ein Objekt dieser Klasse und übergibt den AST zur weiteren Ausführung.

Der `SelectExecutor` enthält Felder für das `select` Element, die `from` Liste und die `where` Bedingung. Um die durch das Durchlaufen der `from` Liste neu vergebenen Bezeichnungen und ihre zugehörigen Daten zu speichern, ist außerdem eine hierarchische Symboltabelle vorhanden. Diese erlaubt es, Namen in aufeinander aufbauenden Gültigkeitsbereichen (Scopes) an Datenobjekte zuzuweisen. Falls es sich bei dem auszuführenden `select` um ein toplevel `select` handelt, wird diese Symboltabelle mit den Namen und Daten aller eNF2 Objekte initialisiert. Anderenfalls (also bei einem Sub-`select`) wird die Symboltabelle des übergeordneten `select`s benutzt, da bei Sub-`select`s auch die Variablenbindungen der umgebenden Anweisung verfügbar sein müssen.

Im nächsten Schritt wird geprüft, ob eine `from` Liste vorhanden ist, d.h. ob eine Schleife ausgeführt werden muss, oder ob das `select` Element nur einmal ausgewertet werden muss.

Das Auswerten des `select` Elements erfolgt in der separaten Methode `evalSelectItem`, da das Element verschachtelt sein kann. Zum Beispiel muss für Tupel diese Methode rekursiv für die einzelnen Felder aufgerufen werden. Abhängig vom Typ des `select` Elements, kann es bei der Auswertung folgende Fälle geben:

Id
: Die Id wird mit Hilfe der aktuellen Symboltabelle aufgelöst und das referenzierte Datenobjekt zurückgeliefert. Dabei kann die Id nicht nur auf toplevel eNF2 Objekte verweisen, sondern auch auf Teilobjekte wie zum Beispiel Listenelemente.

Atomares Literal
: Für das atomare Literal wird ein passendes Datenobjekt (`Enf2Boolean`, `Enf2Char` oder `Enf2Integer`) erzeugt und zurückgegeben.

Liste oder Multimenge
: Es wird ein Datenobjekt des gleichen Typs erzeugt und für jedes Kind im AST `evalSelectitem` aufgerufen. Die Rückgabewerte dieser rekursiven Aufrufe werden zu dem Datenobjekt hinzugefügt und dieses dann zurückgegeben.

Tupel
: Da man zu Tupeln nach dem Erstellen keine weiteren Felder mehr hinzufügen kann, werden in diesem Fall zuerst die Kind-Elemente ausgewertet und in einer Java-Liste gespeichert. Aus dieser Liste wird dann ein Tupel erstellt und zurückgegeben.

Select
: Hier findet die Bearbeitung von Sub-`select`s statt. Für sie wird ein neuer `SelectExecutor` erzeugt, der auf die Symboltabelle des aktuellen `SelectExecutor` verweist. Dann wird der aktuelle Teilbaum des AST durch den neuen

`SelectExecutor` ausgewertet und das Ergebnis der rekursiven Auswertung zurückgegeben.

Für `selects` ohne `from` Liste ist `evalSelectItem` der einzige Schritt, der durchgeführt werden muss.

Ist jedoch eine `from` Liste vorhanden, beginnt die Ausführung einer `select` Anweisung stattdessen mit `fromLoop`. Diese Methode ist für das Durchlaufen der `from` Liste zuständig. Außerdem bestimmt sie den Ergebnistyp der `select` Anweisung. Wenn einmal in der `from` Liste eine eNF2 Liste benutzt wird, dann ist der Ergebnistyp `list` ansonsten `multiset`.

Als erstes wird die, nach dem `in` stehende Id aufgelöst. Jetzt wird geprüft, ob es bereits ein Datenobjekt für das Ergebnis der `select` Anweisung gibt. Wenn nicht, wird geprüft, ob das Datenobjekt zu der Id eine Liste ist. Falls ja, wird als Ergebnisobjekt eine `Enf2List` angelegt. Falls nein, kommt es darauf an, ob wir uns im letzten Element der `from` Liste befinden. Da dann keine Liste mehr im `from` folgen kann, wird als Ergebnisobjekt ein `Enf2Multiset` erzeugt.

Danach wird das Datenobjekt in einer Schleife durchlaufen und bei jedem Durchlauf die Id vor dem `in` in der Symboltabelle an das aktuelle Kindobjekt gebunden.

Wenn es noch weiter Einträge in der `from` Liste gibt, ruft sich `fromLoop` selbst auf, um so eine verschachtelte Schleife zu realisieren. Andernfalls befinden wir uns jetzt in der innersten Schleife und es müssen gegebenenfalls Daten zurückgeliefert werden. Dies hängt davon ab, ob es eine `where` Bedingung gibt. Falls ja, wird mit der aktuellen, durch die `from` Liste oder übergeordneten `select` Anweisungen erweiterten Symboltabelle der boolsche Ausdruck der `where` Bedingung ausgewertet. Ist er `false`, wird der aktuelle Schleifendurchlauf übersprungen. Andernfalls wird jetzt mit der aktuellen Symboltabelle die bereits beschriebene Methode `evalSelectItem` aufgerufen und die Rückgabe an das Ergebnisobjekt angehängt.

Nachdem alle Schleifen in den `fromLoop` Methoden durchlaufen sind, ist das Ergebnisobjekt gefüllt und kann als Ergebnis der `select` Methode zurückgegeben werden.

`dmlStatement` Da auch DML Anweisungen verschachtelt sein können, wurde für sie auch eine spezielle Klasse erstellt: `DmlStatementExecutor`. Sie enthält ebenso wie der `SelectExecutor` eine Symboltabelle für die durch Schleifen entstehenden Variablen. `dmlStatement` erstellt ein Objekt dieser Klasse und übergibt den AST an die Methode `execute`. Diese prüft dann den Typ der Anweisung und führt die jeweils zuständige Methode aus:

`delete`
: Hier wird das zu der Id gehörende Objekt gesucht und gelöscht. Dies kann entweder ein Element einer Liste oder einer Multimenge, oder ein vollständiges eNF2 Objekt sein.

`extend`
: Diese Methode löst zuerst die Id des Objekts auf, das erweitert werden soll. Ist es keine Liste, wird eine Ausnahme geworfen. Andernfalls wird überprüft, ob die übergebene Position (zum Beispiel `after 2`) für dieses Objekt gültig ist. Als nächstes wird zu dem Listenliteral ein Baum von `Enf2Type` Objekten aufgebaut und geprüft, ob die Struktur dieses Baums mit der Struktur des Zielobjekts kompatibel ist. Falls ja, werden die Datenobjekte in das Zielobjekt eingefügt.

`insert`
: Auch hier wird als erstes das Zielobjekt der Einfügung ermittelt. Da `insert` nur für Multimengen verwendet wird, ist hier keine Position zu überprüfen. Zum angegebene Multimengenliteral wird wiederum ein Baum aus Datenobjekten erstellt, der dann auf Kompatibilität mit der Definition des Zielobjekts geprüft wird. Im Erfolgsfall werden die Datenobjekte dann in die Ziel-Multimenge eingefügt.

`update`
: Wiederum wird das Zielobjekt, das durch die Id links des Zuweisungsoperators `:=` bezeichnet wird, gesucht. Dann wird das Literal auf der rechten Seite des Operators in Datenobjekte umgewandelt und die Strukturkompatibilität geprüft. Anschließend wird das Zielobjekt mit den neuen Daten überschrieben.

`for_each`
: Als erstes wird über die Id rechts des `in` Schlüsselwortes, die Liste oder Multimenge bestimmt, über die iteriert werden soll. Diese wird dann durchlaufen und das jeweils aktuelle Element wird in der Symboltabelle an die Id gebunden, die links des `in` Schlüsselwortes steht. Falls eine `where` Klausel vorhanden ist, wird die Bedingung mit der aktuellen Symboltabelle ausgewertet. Ist sie `false`, wird das aktuelle Element übersprungen. Ansonsten wird rekursiv die `execute` Methode mit der im `do` – `end` Block eingeschlossenen DML Anweisung ausgeführt.

Kapitel 3

Was ich mir als Implementierungsgrundlage gewünscht hätte

3.1 Mehrfachvererbung

Die Mehrfachvererbung wäre zum Beispiel bei der Implementierung der `getDefinition` Methode des `Enf2Type` Interface hilfreich gewesen. Um die Forwarding Klassen von Google Collections benutzen zu können, müssen die Implementierungsklassen des `Enf2Type` Interface diese erweitern. Dadurch kann das, für `getDefinition`, notwendige Feld und die Getter-Methode nicht durch eine gemeinsame Basisklasse implementiert werden, sondern muss in jeder Klasse separat implementiert werden.

3.2 Erzeugung von Objekten einer Klasse, die erst zur Laufzeit feststeht

Der HDBL Trainer muss oft zu einer, erst zur Laufzeit feststehenden, Klasse Objekte erzeugen. Dies ist zum Beispiel beim Erstellen eines neuen eNF2 Objekts der Fall, da hierbei erst durch die Eingabe der `create object` Anweisung feststeht für welche Klassen Objekte erzeugt werden müssen. Das ist in Java nur umständlich zu realisieren. Welche Probleme genau dabei auftreten, wird in den folgenden Abschnitten beschrieben.

Möglichkeit, Konstruktoren vorzuschreiben Im Moment ist es in Java nicht möglich, zu erzwingen, dass eine Klasse einen Konstruktor mit einer be-

stimmten Signatur besitzt. In einem Interface kann man keine Konstruktoren deklarieren (Kapitel 9.1.4 Interface Body and Member Declaration, [GJSB05]) und auch statische Methoden sind in Interfaces nicht erlaubt (Kapitel 9.4 Abstract Method Declarations, [GJSB05]).

Da Konstruktoren nicht vererbt werden (Kapitel 8.8 Constructor Declarations, [GJSB05]), ist es auch nicht möglich, den gewünschten Konstruktor in einer Basisklasse zu deklarieren. Selbst wenn Konstruktoren vererbt würden, so wäre diese Vorgehensweise wegen fehlender Mehrfachvererbung immer noch darauf angewiesen, dass die Basisklasse von allen Subklassen benutzt werden kann. Wenn diese jedoch bereits eine andere Klasse erweitert, ist auch das nicht möglich.

Wenn man Konstruktoren für Klassen vorschreiben könnte, wäre das Instanzieren von Objekten zu einer gegebenen `Class<? extends Enf2Type>` einfacher. Denn somit könnte sichergestellt werden, dass jede Klasse, die `Enf2Type` implementiert, einen Konstruktor besitzt, der genau einen `Structure` Parameter hat. Diesen könnte man dann einfach mit `new type(definition)` aufrufen.

Eine Weise, um dies zu ermöglichen, wäre zum Beispiel, dass man Konstruktoren oder statische Methoden in Interfaces deklarieren kann. So könnte man entweder diesen Konstruktor aufrufen oder die statische Methode als Konstruktorersatz benutzen.

Als Ersatz für eine solche Möglichkeit wurde die Erstellung von Instanzen aus `Class<? extends Enf2Type>` Objekten über Reflection realisiert. Hierbei wird über das `Class<? extends Enf2Type>` Objekt der Klasse, für die eine Instanz erzeugt werden soll, ein Konstruktor gesucht, der die Signatur `Structure` hat. Anschließend wird dieser aufgerufen.

Klassen als Objekte erster Klasse Eine andere Möglichkeit, das Erzeugen von Objekte zu einer erst zur Laufzeit bekannten Klasse zu vereinfachen, wäre das Einführen von Klassen als Objekte erster Klasse. Das bedeutet, dass Klassen nicht nur zur Kompilierzeit vorhandene Konstrukte sind, sondern unter anderem auch zur Laufzeit in Variablen gespeichert werden können. So zum Beispiel in dem `type` Feld der `Structure` Klasse.

Somit wäre das Erstellen von Instanzen zur jeweiligen, in `type` gespeicherten Klasse einfacher. Man könnte entweder mit `new type(definition)` den Konstruktor der in `type` gespeicherten Klasse oder mit `type.ofDefinition(definition)` eine statische Methode zur Objekterzeugung aufrufen. So würde das Umständliche Ermitteln von Konstruktoren oder Methoden per Reflection wegfallen.

3.3 Statische Methoden in Interfaces

Wie bereits erwähnt, ist es nicht möglich, statische Methoden in Interfaces zu deklarieren. Dadurch kann man über ein Interface nicht festlegen, dass eine Klasse

eine bestimmte Klassenmethode haben muss. Dies ist nur mit Hilfe einer statischen Methode in einer Superklasse möglich. Dazu ist aber, mangels Mehrfachvererbung, notwendig, dass die Subklassen nicht bereits eine andere Superklasse erweitern. Genau dass aber ist bei den Klassen für die Datenspeicherung im Hauptspeicher der Fall, wodurch diese Möglichkeit ausscheidet. Deshalb musste zum Beispiel eine Methode, die für jede dieser Klassen den eNF2 Namen (zum Beispiel `tuple` für die Klasse `Enf2Tuple` oder `multiset` für `Enf2Multiset`) zurückgibt, anstatt in den Klassen selbst, in der Hilfsklasse `Enf2Util` implementiert werden. Diese unnötige Aufspaltung von Methoden einer Klasse führt zu schwieriger wartbarem Code, da man bei Änderungen immer daran denken muss, dass nicht nur die Klasse selbst, sondern auch die Hilfsklasse geändert werden muss.

3.4 ANTLR

3.4.1 Token Typen

Die Typen der Tokens des Lexers und des Parsers werden von ANTLR als `int`-Konstanten deklariert. Dies führt zu Problemen, wenn die Grammatiken so geändert werden, dass sich die Werte der Token-Typen ändern. Das kann zum Beispiel der Fall sein, wenn man die Reihenfolge der Tokens ändert oder neue Tokens hinzufügt.

Da `static final` Konstanten vom Compiler in die `class` Datei direkt als Wert und nicht als Verweis auf die Konstante einkompiliert werden können (13.4.9 final Fields and Constants [GJSB05], Kapitel 6.4.2 – Eincompilierte Belegungen der Klassenvariablen [Ull09]), ist es möglich, dass sich die Nummerierung der Tokens in den generierten Klassen ändern, aber die in den restlichen Klassen verwendeten Konstanten sich noch auf die alten Werte beziehen. Eine Abhilfe für dieses Problem wäre es, wenn ANTLR die Token-Typen als `enum` (siehe 8.9 Enums [GJSB05]) deklarieren würde. Dann könnte ANTLR Token-Typen hinzufügen oder ihre Reihenfolge ändern, ohne dass diese inkompatibel zu bereits kompiliertem Code werden (13.4.26 Evolution of Enums [GJSB05]).

3.4.2 Grammatiken

Codierung Es ist nicht möglich für eine Grammatikdatei die verwendete Codierung festzulegen (zum Beispiel ASCII oder ISO-8859-1). ANTLR setzt immer voraus, dass die Grammatik ASCII codiert ist. Nicht-ASCII Zeichen werden in den Grammatiken zur Zeit nur über den Umweg von Unicode Escapesequenzen unterstützt. Mit diesen ist es möglich, über die Notation `\uXXXX` ein Unicode Zeichen aus der Basic Multilingual Plane zu benutzen (siehe `http://www.unicode.org`). Immer wenn solche Zeichen in einer Grammatik auftauchen, wird diese deshalb unnötig kompliziert und unübersichtlich. Zum Beispiel sähe ein Ausschnitt einer Grammatik, die die Umlaute äöüÄÖÜß erkennt fol-

gendermaßen aus:

```
UMLAUT: '\u00e4' | '\u00f6' | '\u00fc' |
        '\u00c4' | '\u00d6' | '\u00dc' | '\u00df';
```

Mit Unterstützung einer Codierung, die diese Zeichen direkt unterstützt (zum Beispiel Unicode) ergäbe sich folgende, deutlich einfacher zu lesende Version:

```
UMLAUT: ä'' | ö'' | ü'' | Ä'' | Ö'' | Ü'' | ß'';
```

Außerdem ist es bei der Definition von Regeln, die Zeichenketten enthalten, durch die Unicode Escapesequenzen notwendig, die jeweiligen Zeichen im Unicode Standard nachzuschlagen, auch wenn man sie eigentlich direkt eingeben könnte.

Zeichenmengen Eine weitere Verbesserung wäre die Möglichkeit, Zeichen, die nicht erkannt werden sollen, explizit als auszulassende Zeichen angeben zu können. Bislang können nur "positive" Zeichenmengen angegeben werden. Um Zeichen auszuschließen muss man die Zeichenmenge, aus der man Zeichen auslassen möchte, in mehrere Teil-Zeichenmengen aufspalten und diese getrennt angeben.

3.4.3 Fehlerbehandlung

ANTLR verwendet zur Behandlung von Fehlern im Eingabetext standardmäßig eine Single-Token-Insertion-and-Deletion Strategie. Dabei versucht ANTLR durch Einfügen eines Tokens ein fehlendes Token und durch Weglassen eines Tokens ein zuviel vorhandenes Token zu kompensieren. Dies sähe für Listenliterale beispielsweise so aus:

Token-Deletion Wenn die Eingabe `<1, 2>>` lautet, würde ANTLR beim Parsen des letzten `>` bemerken, dass an dieser Stelle eigentlich das Eingabeende sein müsste. Da direkt an der nächsten Stelle wirklich das Eingabeende kommt nimmt ANTLR an, dass das `>` ein überschüssiges Zeichen ist und überspringt es.

Token-Insertion Bei der Eingabe `<1, 2` würde ANTLR beim Erreichen des Eingabeendes bemerken, dass noch ein `>` fehlt. Da auf dieses Zeichen das Eingabeende folgen kann, verhält sich ANTLR so, als ob es eingegeben worden wäre.

Eine ausführlichere Beschreibung dieser zwei Verfahren befindet sich in Kapitel 10.7 in [Par07].

Diese automatischen Fehlerbehandlung ist für den hauptsächlichen Einsatzzweck der generierten Lexer und Parser – das Erkennen von Quelldateien –

gut, da so meist mehrere Fehler auf einmal gefunden und korrigiert werden können, ohne dass es nötig ist, für jeden Fehler den aufwändigen Lexing- und Parsing-Durchlauf anzustossen. Für den Einsatz als interaktiver Interpreter in einer Lernumgebung, ist diese Art der Fehlerbehandlung jedoch ungeeignet, da hierbei nur kurze Quelltexte erkannt werden müssen. Daher ist die Zeitersparnis durch das Vermeiden mehrerer Durchläufe gering. Außerdem kann es durch die automatische Fehlerkorrektur zu Folgefehlern kommen, wenn Fehler gemeldet werden, die nur durch vorhergehende Fehler verursacht werden. Diese Folgefehler sind für Studenten, die HDBL lernen, verwirrend, da sie sich nicht sicher sein können, ob ein gemeldeter Fehler ein tatsächlicher Fehler in ihrer Eingabe ist, oder ob es sich nur um einen Folgefehler der automatischen Fehlerkorrektur handelt. Aus diesem Grund sollte der HDBL Trainer keine automatische Fehlerkorrektur durchführen, sondern beim ersten Fehler im Lexer oder Parser die Erkennung abbrechen und dem Benutzer den Fehler melden.

Diese Fehlerbehandlungsstrategie lässt sich nicht einfach umkonfigurieren. Um die Strategie zu ändern, ist es außerdem notwendig, mehrere Methoden zu überschreiben. Im einzelnen sind folgende Schritte durchzuführen:

Überschreiben von `recoverFromMismatchedToken`

Die Methode `recoverFromMismatchedToken` ist dafür zuständig, Fehler durch fehlende oder zuviel vorhandene Tokens automatisch zu korrigieren. Die überschreibende Methode umgeht dieses Verhalten nun, indem sie immer die gleiche Exception (`MismatchedTokenException`) wirft, die von der ursprünglichen Methode geworfen worden wäre, wenn keine automatische Fehlerkorrektur möglich gewesen wäre. Dadurch erhält der Aufrufer bei einem solchen Fehler in jedem Fall die Möglichkeit, diesen durch Behandlung dieser Exception zu bearbeiten.

Code für Exceptionhandling im Parser festlegen

Der Code, den ANTLR benutzt, um Exceptions in Regeln abzufangen und zu behandeln, lässt sich in der Grammatik mit einem `@rulecatch{}` Block angeben. Der innerhalb der geschweiften Klammern angegebene Code ersetzt den von ANTLR standardmäßig generierten Code zur Ausnahmebehandlung in Regeln. Dieser würde den Fehler auf dem Standardfehlerausgabekanal ausgeben und danach versuchen, diesen automatisch zu beheben. Der neue Code im `@rulecatch{}` Block wirft stattdessen erneut die gefangene Ausnahme und ermöglicht so dem Aufrufer, diese zu abzufangen.

Ausnahmebehandlung im Lexer

Da sich im Lexer der Code für die Ausnahmebehandlung nicht mit einem `@rulecatch{}` Block angeben lässt, ist hier eine andere Vorgehensweise nötig. Dabei wird die `displayRecognitionError(String[] tokenNames, RecognitionException e)`

Methode überschrieben, die normalerweise die Ausnahme auf dem Standardfehlerausgabekanal ausgibt. Da diese Methode nicht deklariert, dass sie die Ausnahme `RecognitionException` wirft, können die aufgetretenen Exceptions nicht einfach mit `throw` weitergegeben werden. Deshalb wurde eine neue Exception `LexerException` angelegt, die in einem Feld die ursprüngliche `RecognitionException` speichert. Da diese neue Ausnahme von RuntimeException abgeleitet wurde, ist sie eine ungeprüfte Ausnahme (11.5 Exception Hierarchy [GJSB05]) und kann somit, ohne im Methodenkopf deklariert zu sein, geworfen werden. Durch die Verwendung einer ungeprüften Ausnahme wird durch den Kompiler nicht mehr überprüft, ob die geworfene Ausnahme auch behandelt wird. Da die betreffende Ausnahme aber nur an einer Stelle im Code behandelt werden muss, ist die dadurch entstehende Unsicherheit vernachlässigbar.

Teil II

Benutzerhandbuch

Kapitel 4

Systemvoraussetzungen und Installation

Der HDBL Trainer setzt die Java Laufzeitumgebung in Version 6 (Java™ SE Runtime Environment 6) oder höher voraus. Da sich der HDBL Trainer zusammen mit allen benötigten Bibliotheken in einer `.JAR` Datei befindet, ist keine Installation notwendig.

Jedoch muss das Home-Verzeichnis des Benutzers für Schreibzugriffe freigegeben sein, damit dort die History der eingegebenen HDBL Anweisungen gespeichert werden kann.

Kapitel 5

Bedienung

Die Bedienung des HDBL Trainers findet über eine graphische Benutzeroberfläche statt. Diese besteht aus dem Anweisungstextfeld rechts oben, dem Ergebnistextfeld rechts unten, der History auf der linken Seite und einer Menüleiste.

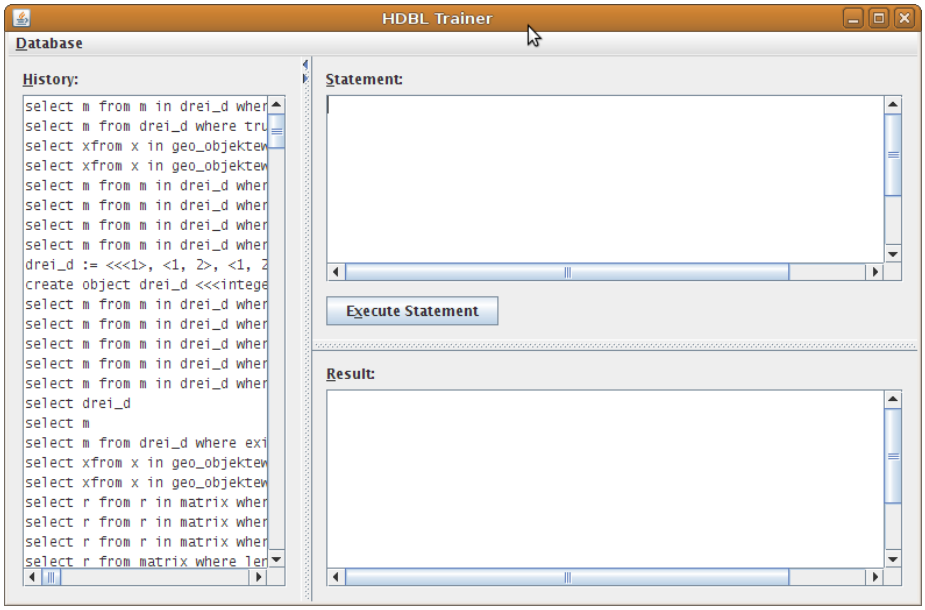


Abbildung 5.1: Benutzeroberfläche

5.1 Kommandozeilenparameter

Falls ein Kommandozeilenparameter angegeben wird, wird dieser als Dateiname einer Datenbank interpretiert. Beim Programmstart wird diese dann automa-

tisch geladen.

5.2 Anweisungstextfeld

Ein Textfeld, das sogenannte Anweisungsfeld, nimmt die HDBL Anweisungen entgegen, die vom HDBL Trainer ausgeführt werden. Drückt man in diesem Feld die Return Taste, wird eine neue Zeile geöffnet. Um die eingegebene Anweisung auszuführen, kann man entweder die Schaltfläche unterhalb des Eingabefelds benutzen oder im Eingabefeld einfach Ctrl-Enter drücken.

5.3 Ergebnistextfeld

Das Ergebnis der Anweisung wird im Erfolgsfall im Ausgabefeld rechts unten angezeigt. Sollte es bei der Ausführung zu einem Fehler kommen, wird dieser in einem separaten Dialog angezeigt. Falls der Grund des Fehlers eine inkorrekte HDBL Anweisung ist, wird im Fehlerdialog die Stelle in der Anweisung hervorgehoben, an der sich der Fehler befindet. Außerdem wird angezeigt, was stattdessen an dieser Stelle erwartet wird. Wenn die Anweisung an sich keinen Fehler enthält, aber aufgrund des Inhalts der Datenbank nicht ausgeführt werden kann (zum Beispiel beim Anlegen eines Objekts mit einem schon genutzten Namen), wird dem Benutzer der genaue Fehlergrund mitgeteilt.

5.4 History

Damit mehrmals benötigte Anweisungen nicht immer wieder eingegeben werden müssen, gibt es eine History. Diese befindet sich am linken Rand und besteht aus einer umgekehrt chronologisch sortierten Liste von ausgeführten Anweisungen. Es wird jede ausgeführte Anweisung gespeichert, unabhängig davon, ob sie erfolgreich war oder nicht. Damit können auch Anweisungen wiederholt werden, die aufgrund des aktuellen Datenbankzustands nicht ausgeführt werden konnten. Dies ist zum Beispiel dann hilfreich, wenn man eine Abfrage durchführen möchte, jedoch vergessen hat, vorher ein dafür benötigtes Objekt anzulegen.

In der History sind folgende Tastenkürzel definiert:

`Enter`
: Der aktuell selektierte Eintrag wird im Anweisungstextfeld eingetragen und dieses erhält den Eingabefokus. Achtung! Der bisherige Text wird ohne Nachfrage überschrieben.

`Ctrl-Enter`
: Der selektierte Eintrag wird in das Anweisungsfeld eingetragen und automatisch ausgeführt. Der Eingabefokus verbleibt hierbei in der History.

Damit kann eine Reihe von Statements aus der History schnell ausgeführt werden. Wie bei `Enter` wird bereits bestehender Text ohne Nachfrage ersetzt.

`Entf`
: Mit dieser Taste werden Einträge gelöscht.

Die History wird beim Programmstart automatisch geladen und beim Verlassen des Programms automatisch gespeichert.

5.5 Menü

Im Menü kann man eine eNF2 Datenbank laden und die aktuelle Datenbank speichern. Außerdem gibt es einen Menüpunkt zum Beenden des HDBL Trainers.

5.6 Fehlerdialog

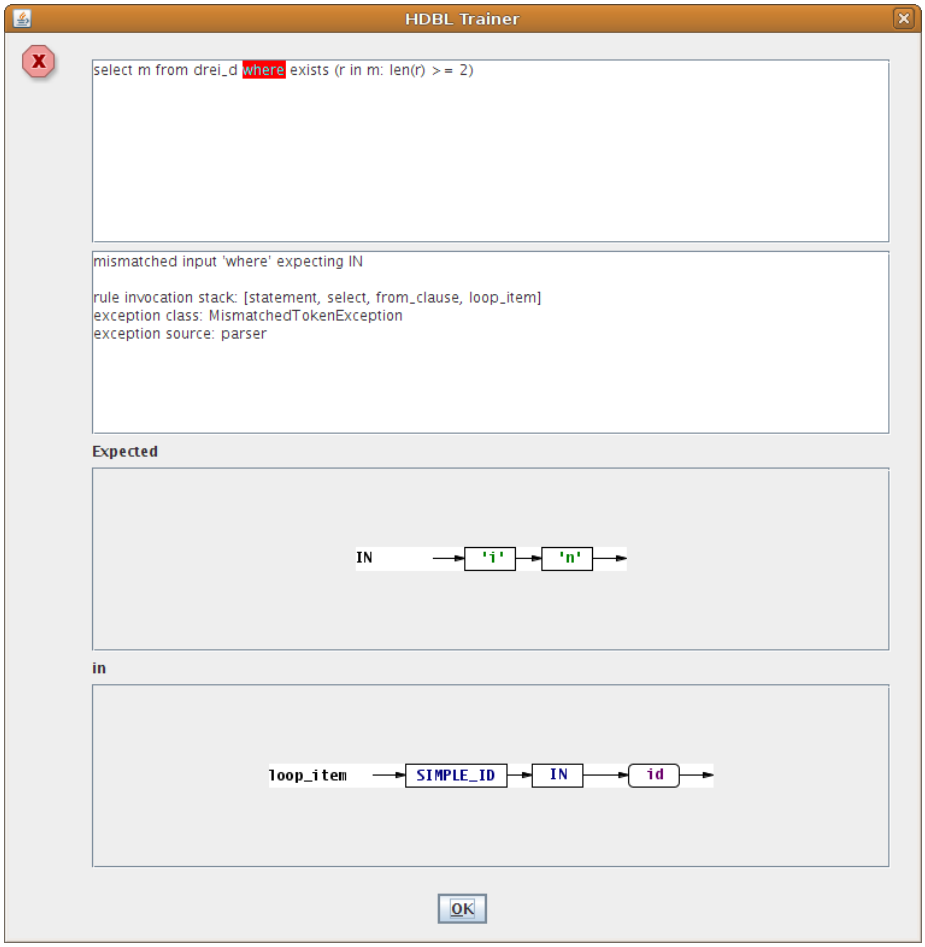


Abbildung 5.2: Fehlerdialog

Falls die eingegebene HDBL Anweisung einen Fehler aufweist, erscheint ein Fehlerdialog, in dem der aufgetretene Fehler beschrieben wird. Es wird die Stelle in rot hervorgehoben, an der der Fehler aufgetreten ist. Zudem werden Syntaxdiagramme angezeigt, die erläutern, wie die erwartete Eingabe aussehen müsste. Beim Verlassen des Dialogs wird der Cursor im Anweisungstextfeld platziert und die fehlerhafte Stelle der eingegebenen Anweisung markiert. So kann der Fehler einfach korrigiert werden.

Kapitel 6

Unterstützte HDBL Syntax

6.1 Allgemeine Sprachkonstrukte

6.1.1 Datentypen und Literale

Der HDBL Trainer unterstützt als atomare Datentypen boolscher Wert (`boolean`), Zeichenkette (`char`) und Ganzzahl (`integer`). Als strukturierte Typen werden Liste (`list`), Multimenge (`multiset`) und Tupel (`tuple`) unterstützt.

Literale der atomaren Datentypen werden in folgendem Format angegeben:

Boolscher Wert
: Die boolschen Literale sind `true` und `false`.

Zeichenkette
: Zeichenkettenliterale werden in einfache Anführungszeichen eingeschlossen. Innerhalb dieser Anführungszeichen sind die Unicode Zeichen in den Bereichen `\u0020` bis `\u0026`, `\u0028` bis `\u007E` und `\u00A1` bis `\u00FF` erlaubt. Diese Bereiche enthalten unter anderem auch die deutschen Umlaute und das scharfe s (ß). Einfache Anführungszeichen sind innerhalb einer Zeichenkette nicht erlaubt. Beispiel für eine gültige Zeichenkette: `'Hallo, Herr Maier'`

Ganzzahl
: Ganzzahlliterale bestehen aus einer mindestens einstelligen Folge aus Ziffern im Bereich von null bis neun. Negative Literale sind derzeit noch nicht erlaubt.

Literale der strukturierte Datentypen sind in das jeweilige öffnende und schließende Typkennzeichen eingeschlossen. Dazwischen befinden sich, durch Kommata abgetrennt, die Literale der Elemente (bei Listen und Multimengen) oder der Felder (bei Tupeln). Listen und Multimengen können leer sein; dann folgt

auf das öffnende Typkennzeichen direkt das schließende. Leere Tupel sind nicht möglich. Sie müssen mindestens ein Feld besitzen.

Die Typkennzeichen sind:

Strukturierter Typ	öffnendes Kennzeichen	schließendes Kennzeichen
Liste	<	>
Multimenge	{	}
Tupel	[	]

Tabelle 6.1: Typkennzeichen

6.1.2 Datenausdruck

Ein Datenausdruck ist eine spezielle Art von Ausdruck, der in boolschen Ausdrücken benutzt werden kann und eNF2 Datenobjekte zurückliefert. Ein Datenausdruck kann eine Id, ein Literal oder das Ergebnis eines Funktionsaufrufs sein.

6.1.3 Schleifenausdruck

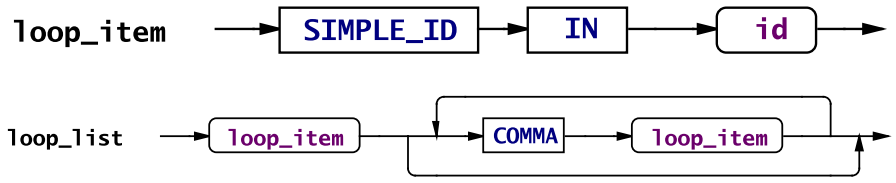


Abbildung 6.1: Schleifenausdruck

Schleifenelement Schleifenelemente werden in `for_each` Anweisungen und in Schleifenlisten benutzt, um anzugeben, über welche Listen oder Multimengen die Anweisung iterieren soll. Der Bezeichner `SIMPLE_ID` ist dabei die Schleifenvariable. `id` gibt den Bezeichner der Liste oder der Multimenge an über die iteriert werden soll. Für jeden Durchlauf durch die Schleife, wird das aktuelle Element aus `id` der Variablen `SIMPLE_ID` zugewiesen und die weiteren Anweisungen ausgeführt.

Für `SIMPLE_ID` können auch Bezeichner verwendet werden, die bereits vergeben sind. Diese werden dann innerhalb der Schleife von `SIMPLE_ID` verdeckt und sind dort nicht mehr zugreifbar. Wenn die Schleife über ein eNF2 Objekt auf oberster Ebene läuft (d.h. `id` besteht nicht aus mehreren, durch Punkt abgetrennten Teilen), kann für die Schleifenvariable sogar der gleiche Bezeichner

gewählt werden. Dann sind in der Schleifenanweisung jedoch nur noch die einzelnen Elemente der Liste oder der Multimenge zugreifbar und nicht mehr das ganze Objekt.

Schleifenliste Schleifenlisten kommen in der `from` Klausel in `select` Anweisungen und in Prädikaten zum Einsatz. Mit ihnen können mehrere Schleifenelemente, durch Kommata getrennt, angegeben werden. Die Schleifenelement werden dann in verschachtelten Schleifen von links nach rechts abgearbeitet. Dies bedeutet, dass das zuerst genannte Schleifenelement die äußerste Schleife, und das zuletzt genannte die innerste Schleife darstellt.

6.1.4 Boolscher Ausdruck

Boolsche Operatoren Es werden die boolschen Operatoren `and`, `or` und `not` unterstützt. Die Auswertung der Operatoren `and` und `or` erfolgt dabei im Kurzschlussverfahren. Das heißt, dass zuerst der linke Operand ausgwertet wird. Nur wenn dann das Ergebnis noch nicht feststeht, wird auch der rechte Operator ausgewertet. Bei `and` wird also der rechte Operator nur ausgewertet, falls der linke `true` ergibt. Bei `or` wird der rechte Operator nur ausgewertet, falls der linke `false` ergibt. Dies kann zum Beispiel dazu benutzt werden, um bei Listen den Inhalt des zweiten Elements zu prüfen: `len(liste) >= 2 and liste.2 = 'x'` Durch die Kurzschlussauswertung wird so verhindert, dass die Auswertung mit einem Fehler abbricht wenn `liste` weniger als zwei Elemente hat.

Vergleichsoperatoren Um in boolschen Ausdrücken Objekte auf Gleichheit oder Ungleichheit untersuchen zu können, dienen die Operatoren `=` und `!=`. Sie können auf Objekte beliebigen Typs angewandt werden. Für die Operatoren `<=` und `>=` müssen die Operanden vom Typ `integer` sein.

Diese Operatoren können auf jeden Datenausdruck angewandt werden.

Prädikate

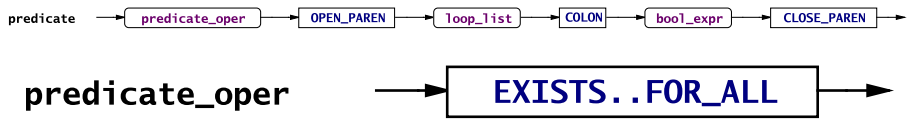


Abbildung 6.2: Prädikate

Mit Hilfe von Prädikaten können mehrere Datenobjekte auf Erfüllung eines boolschen Ausdrucks untersucht werden. Bei `for_all` ist das Prädikat genau dann wahr wenn der boolsche Ausdruck für alle Datenobjekte wahr ist. Bei

`exists` genügt es, wenn der boolsche Ausdruck für ein Datenobjekt wahr ist, damit das gesamte Prädikat wahr ist. Wenn er für keines der Datenobjekte wahr ist, dann ist `exists` falsch. Mit `loop_list` lässt sich festlegen, über welche Datenobjekte iteriert werden soll. `bool_expr` ist ein beliebiger boolscher Ausdruck.

Vorrangregeln der Operatoren Höchste Priorität haben die Vergleichsoperatoren und Prädikate. Sie werden als erstes ausgewertet. Dann folgen die boolschen Operatoren. Sie werden in der Reihenfolge `not`, `and` und `or` ausgewertet. Um eine andere Auswertungsreihenfolge zu erzwingen, kann man Teilausdrücke in Klammern einschließen. Zum Beispiel wird im Ausdruck `a and b or c` der Teilausdruck `a and b` zuerst ausgewertet. Wenn als erstes `b or c` ausgewertet werden soll, muss `a and (b or c)` eingegeben werden.

6.1.5 Funktionen

`len` Die Funktion `len` ermittelt die Größe eines Objekts. Sie erwartet einen Datenausdruck als Parameter und liefert einen `integer` zurück. Für Listen und Zeichenketten liefert `len` die Länge der Liste bzw. der Zeichenkette zurück. Für Multimengen wird die Kardinalität zurückgegeben. Dies bedeutet, dass mehrfach vorhandene Elemente auch mehrfach gezählt werden. Die Verwendung von `len` mit anderen Datentypen wird nicht unterstützt und führt zu einer Fehlermeldung.

6.1.6 `where` Klausel

Abbildung 6.3: `where` Klausel

Die `where` Klausel findet in `for_each` und `select` Anweisungen Verwendung. Mit ihr lässt sich ein boolscher Ausdruck (siehe Punkt 6.1.4, S. 49) angeben, der bestimmt, welche der Schleifendurchläufe dieser Anweisungen wirklich durchgeführt werden sollen. Nur wenn die Auswertung des boolschen Ausdrucks, im Kontext der durch die Schleife gebundenen Variablen, `true` ergibt, wird der jeweilige Schleifendurchlauf ausgeführt.

6.2 create object

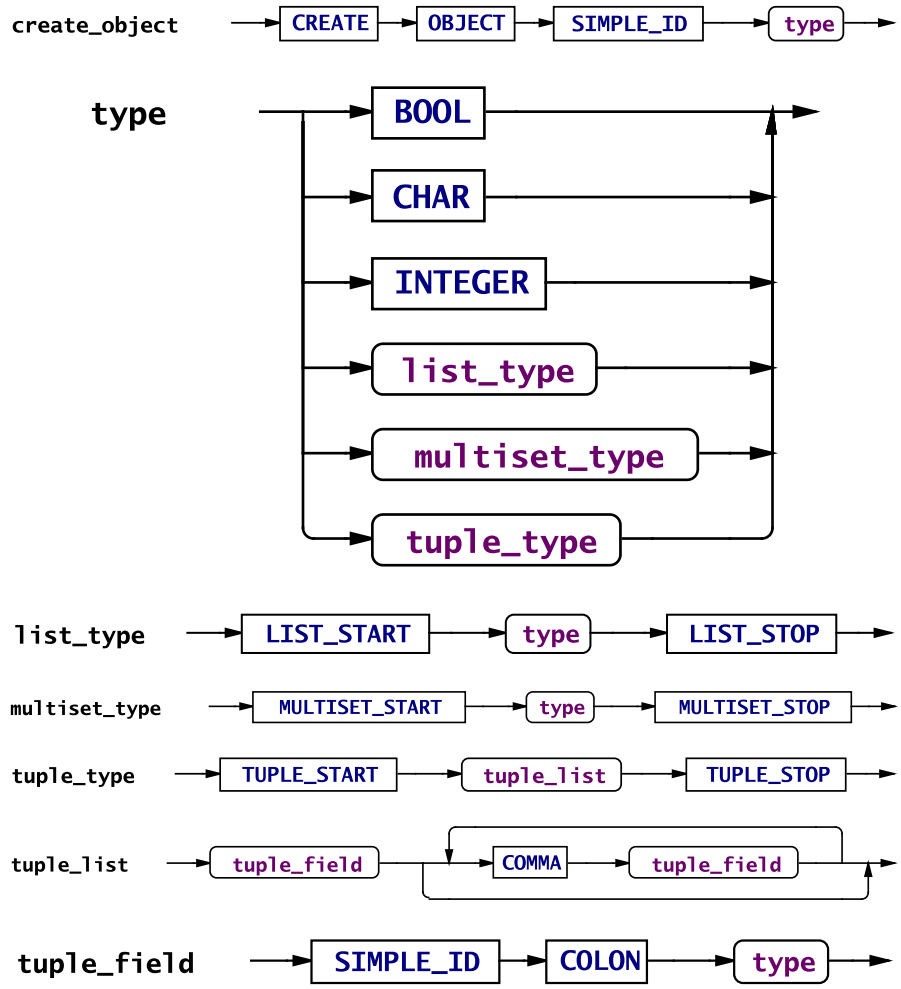


Abbildung 6.4: `create object`

Mit der `create object` Anweisung werden neue eNF2 Objekte angelegt. Diese Anweisung benötigt als Parameter den Namen und den Typ, den das neue Objekt haben soll. Der Typ ist dabei entweder ein atomarer Typ, wie `boolean`, `integer` und `char` oder einer der strukturierten Typen Liste, Multimenge und Tupel. Diese strukturierten Typen werden durch ein öffnendes und schließendes Typkennzeichen (6.1, S. 48) angegeben. Eingeschlossen zwischen den Typkennzeichen wird der enthaltene Typ angegeben.

6.3 Describe Object

Abbildung 6.5: Describe Object

Die Definition eines Objekts lässt sich mit der `describe object` Anweisung abfragen. Der Parameter gibt den Namen des Objekts an, zu dem man sich die Definition ausgeben lassen möchte. Falls das Objekt mit dem angegebenen Namen nicht existiert, wird eine Fehlermeldung ausgegeben. Die Definition wird im gleichen Format ausgegeben, wie man sie bei einer `create object` Anweisung angeben müsste, um das gleiche Objekt zu erstellen.

6.4 `select`

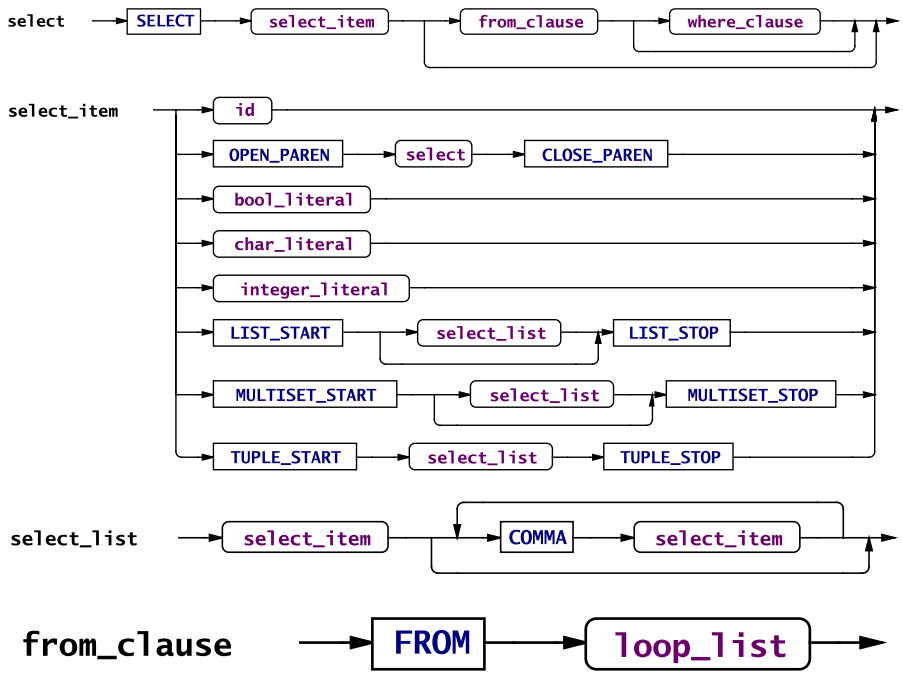


Abbildung 6.6: `select`

Mit einer `select` Anweisung lassen sich Daten aus den eNF2 Objekten abfragen. Sie bestehen mindestens aus dem Schlüsselwort `select` und einem `select_item`.

Zusätzlich kann eine `from` Klausel angegeben werden, die genauer bestimmt woher die Daten stammen sollen. Wenn eine solche angegeben ist, dann kann auch eine `where` Bedingung angegeben werden, die Datenmenge einschränken kann.

6.4.1 `select_item`

Das `select_item` bestimmt welche Daten selektiert werden sollen. Es besteht aus folgenden Elementen:

`id` Es wird das Objekt zu dieser Id selektiert.

`select` Mit einem `select` im `select_item` lassen sich Sub-Selects realisieren. Der Sub-Select wird mit den aktuellen, gegebenenfalls durch die `from` Klausel erweiterten, Variablenbindungen ausgeführt. Das gesamte Ergebnis des Sub-Select wird als Ergebnis des `select_item` zurückgegeben.

Boolscher Wert, Zeichenkette oder Ganzzahl Diese geben einfach den angegebenen Wert zurück.

Liste, Multimenge oder Tupel Es wird ein Objekt gleichen Typs zurückgegeben, wobei jedes Element wiederum ein `select_item` sein kann. So lassen sich verschachtelte Ergebnisse erzeugen.

6.4.2 `from` Klausel

Die `from` Klausel besteht aus dem Schlüsselwort `from` und einer Schleifenliste (Punkt 6.1.3, S. 48). Diese gibt an, über welche Listen oder Multimengen die `select` Anweisung iterieren soll. Falls keine `from` Klausel angegeben wird, findet keine Iteration statt und es wird einfach einmal das `select_item` zurückgegeben. Andernfalls wird der Rückgabetyp durch die Schleifenliste bestimmt. Wenn in ihr mindestens eine Liste benutzt wird, so ist der Ergebnistyp `list`, anderfalls `multiset`.

6.4.3 `where` Klausel

Durch Angabe der optionalen `where` Klausel (Punkt 6.1.6, S. 50) kann das Ergebnis gefiltert werden. Nur Daten für die die `where` Klausel `true` ergibt, werden ausgegeben. Ohne diese Klausel werden alle Daten ausgegeben.

6.5 DML Anweisungen

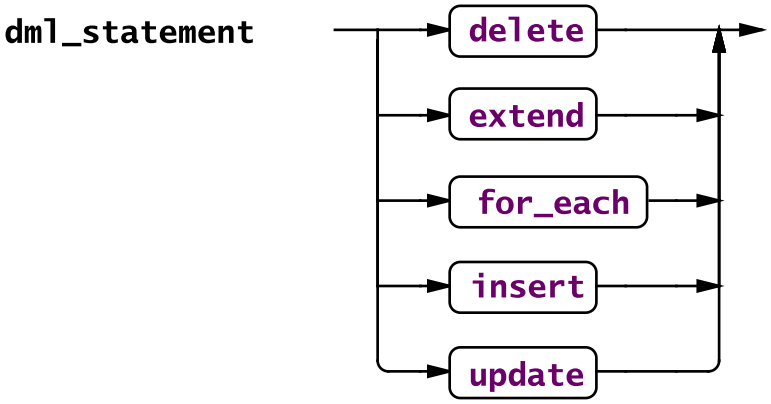


Abbildung 6.7: DML Anweisung

Mit den DML Anweisungen lassen sich die Nutzdaten von Objekten bearbeiten. Mit `delete` können Daten gelöscht werden, `extend` und `insert` dienen zum Hinzufügen von Daten und `update` aktualisiert Daten. `for_each` zählt auch zu den DML Anweisungen, da mit dieser Anweisung eine andere DML Anweisung mehrfach ausgeführt werden kann.

Falls bei der Aktualisierungsanweisung `update` oder einer der Hinzufügeanweisungen `extend` und `insert` das angegebene Literal nicht mit dem Zielobjekt kompatibel ist, wird ein Fehler gemeldet und die Anweisung wird nicht ausgeführt. Das bedeutet, dass bei verschachtelten Literalen nicht versucht wird, die Anweisung bis zum Auftreten eines Fehlers teilweise auszuführen. Dadurch ist sichergestellt, dass eine solche Anweisung atomar ausgeführt wird.

6.5.1 `delete`

Abbildung 6.8: Delete

Diese Anweisung löscht das durch `id` angegebene Objekt. Das Objekt muss entweder ein toplevel Objekt, oder in einer Liste oder Multimenge enthalten sein.

6.5.2 `extend`

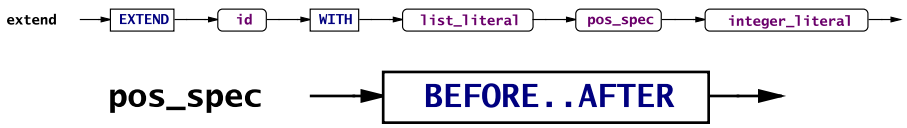


Abbildung 6.9: Extend

Die `extend` Anweisung dient zum Erweitern einer Liste um ein Listen-Literal. Es werden vier Parameter erwartet: eine Id (`id`), ein Listen-Literal (`list_literal`), ein Positionsspezifizierer (`pos_spec`) und ein Integer-Literal (`integer_literal`). Die Id bezeichnet die Liste, die mit dem Listen-Literal erweitert werden soll. Der Positionsspezifizierer `before` oder `after` gibt zusammen mit dem Integer-Literal an, an welcher Stelle die Liste erweitert werden soll. Die Nummerierung der Elemente einer Liste ist eins-basiert. Das heißt, dass die Elemente von eins bis zur Anzahl der Elemente durchnummeriert werden. Mit dem Positionsspezifizierer kann nun festgelegt werden, ob das Listen-Literal vor oder nach einem vorhandenen Element eingehängt werden soll.

6.5.3 `insert`

Abbildung 6.10: Insert

Die `insert` Anweisung dient zum Erweitern einer Multimenge mit einem Multimengen-Literal. `insert` erwartet als ersten Parameter die Id (`id`) einer Multimenge und als zweiten Parameter das Multimengen-Literal (`multiset_literal`), mit dem die Multimenge erweitert werden soll. Da Multimengen unsortiert sind, kann hier keine Position angegeben werden.

6.5.4 `update`

Abbildung 6.11: Update

Mit dem Zuweisungsoperator `:=` können Objekten neue Werte zugewiesen werden. Auf der linken Seite des Zuweisungsoperators muss eine Id (`id`) angegeben

werden, auf der rechten Seite ein Literal, das in seiner Struktur dem Objekt entspricht, das durch `id` referenziert wird.

6.5.5 `for_Each`

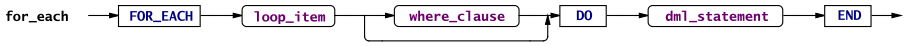


Abbildung 6.12: for_Each

Mit der `for_each` Anweisung wird eine andere DML Anweisung (auch nochmals eine `for_each` Anweisung) in einer Schleife ausgeführt. Über welche Daten die Schleife iterieren soll, wird mit einem Schleifenelement (Punkt 6.1.3, S. 48) angegeben. Optional kann eine `where` Klausel (Punkt 6.1.6, S. 50) angegeben werden, um die auszuführenden Schleifendurchläufe einzuschränken. Ohne `where` Klausel werden alle Schleifendurchläufe ausgeführt.

Teil III

Anhang

Kapitel 7

Hinweise zum Erstellen des HDBL Trainers

Für das Erstellen des HDBL Trainers ist ein Java 6 SDK (zum Beispiel von Sun `http://java.sun.com`) notwendig. Außerdem wird das Build Tool Ant des Apache Projekts (`http://ant.apache.org/`) verwendet. Die Programme ANTLR zum Erstellen des Lexers und Parsers und ANTLRWorks zum Erstellen der Syntaxdiagramme befinden sich bereits im Unterverzeichnis `tools/` und müssen nicht mehr installiert werden.

Die von Ant benutzte Definition des Erstellungsvorgangs befindet sich in der Datei `build.xml`. Diese enthält folgende Targets:

`clean`
: Wenn man alle kompilierten `.class` Dateien und die Syntaxdiagramme löschen möchte (zum Beispiel weil eine Regel aus der Grammatik gelöscht wurde), muss dieses Target aufrufen werden.

`compile`
: Hiermit wird der Kompiliervorgang gestartet. Die erzeugten `.class` Dateien werden im Verzeichnis `bin/` abgelegt.

`generateRecognizer`
: Dieses Target erzeugt mit Hilfe von ANTLR den Lexer und Parser zur Erkennung von HDBL.

  Zuvor wird das Target `clean` aufgerufen, damit bei einem anschließenden Kompilieren alle Dateien neu übersetzt werden. Dies ist notwendig, da sich durch den Aufruf von ANTLR unter Umständen die Konstanten für die Token Typen ändern können und der bereits kompilierte Code sonst weiterhin die alten Werte benutzen würde.

  Anschließend wird auch das Target `syntaxDiagrams` aufgerufen, um die Syntaxdiagramme zu aktualisieren.

`jar`
: Zum einfacheren Weitergeben des kompilierten HDBL Trainers, kann mit dem Target `jar` ein `.JAR` Archiv erstellt werden, das die kompilierten `.class` Dateien, die Syntaxdiagramme und alle benötigten Bibliotheken enthält.

`javadoc`
: Um aus den Javadoc Kommentaren im Code eine Dokumentation im HTML Format zu erzeugen, kann dieses Target benutzt werden. Die Dokumentation wird im Verzeichnis `doc/` gespeichert.

`syntaxDiagrams`
: Die in den Fehlermeldungsdialogen angezeigten Syntaxdiagramme werden durch dieses Target generiert. Dabei werden Fehlermeldungen angezeigt, dass die Datei `HdblBaseLexer.tokens` nicht gefunden werden kann. Diese Fehlermeldungen können ignoriert werden, da die Datei zum Erstellen der Diagramme nicht notwendig ist. Sie werden trotz der Fehlermeldung korrekt generiert. Die fertigen Diagramme finden sich im Verzeichnis `img/`.

Kapitel 8

Erstellungserklärung

Name: Dennis Christoph Benzinger Matrikelnummer: 493094

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

D. Benzinger