Master Thesis
June 2006

Jakob T. Andersen *jta@cs.aau.dk*

Rune D. Hammerskov *rdh@cs.aau.dk*

Lars H. Nielsen *dengmao@cs.aau.dk*

Faculty of Engineering and Science

Aalborg University

Department of Computer Science

Title:
Amigo

Subtitle:
An Object Relational Query Language

Project period:
Dat6
2006

Project group:
d632a

Group members:
Rune D. Hammerskov
Jakob T. Andersen
Lars H. Nielsen

Supervisor:
Lone Leth Thomsen

Copies: 7
Pages: 123

Abstract:

Amigo is a language for querying relational databases from an object oriented setting. Traditionally, querying relational data from object oriented programming languages entails using SQL queries embedded in strings. This approach is not optimal, as queries are neither syntax nor type checked until they are sent to the database at runtime, with the result that fatal errors could occur during execution. Amigo addresses this problem by providing syntax and type checking of its queries. Furthermore, Amigo is designed with intuitiveness and ease of use in mind. Amigo queries are expressed using the concept of filters that was established in the previous project Language Integrated Persistence. This concept is further developed and expanded to accommodate the language features of Amigo.

Rune D. Hammerskov | Jakob T. Andersen | Lars H. Nielsen

Preface

Prerequisites

This report is aimed at software engineers. The reader should have a basic knowledge of both object oriented programming and relational databases. As the report deals with language development the reader should have some knowledge about this as well.

Reading notes

The report consists of nine chapters and three appendices. The first chapter introduces the problem that this report tries to solve. Chapter two consists of an analysis of existing technologies and discussion about the features that the Amigo language should support. Chapter two is summarised in the last section of the chapter. Chapter three describes the design and development process. Chapter three is also summarised in the last section of the chapter. In chapter three an example scenario is presented which is used throughout the rest of the report. Chapter four describes the implementation, and is summarised in the last section of the chapter. Chapter five describes the test of the Amigo language, both a validation and technical test. In this chapter, the a summary is present as well. Chapter six describes related work. Chapter seven is an evaluation of the report, process, and Amigo language. Chapter eight, as a direct continuation of chapter seven, describes future work which could be done in the development of Amigo. Chapter nine is a short conclusion concerning the report, process, tools, and Amigo language.
Throughout this report, an extensive number of abbreviations are used. When a new concept is introduced, for which an abbreviation is applicable, the full name is presented first, and then the abbreviation of that concept follows in parentheses. For instance, Language Integrated Query (LIP).

Typography

There are two types of code examples in this report. The first we call Listings, and this type is reserved for larger and more complete examples. Listings look like this:

Listing 1: Test

```
1 This is a test.
```

In the back of the a report there is a complete list of all listings.
The second type of examples is reserved for smaller examples which often only portray as small part of some code. These examples look like this:

```
This is a test.
```

Tools

In the course of this project, we have made use of several tools. This report is written using Latex and the Latex package memoir. The Amigo compiler is implemented in C# and it runs on both the .NET platform as well as the Mono platform. The integrated development environments used for the implementation is SharpDevelop and Monodevelop.

Compiler & Source code

The implemented Amigo compiler and it's source code, is included on a CD-ROM which can be found in the back of this report. Furthermore, an electronic version of the report is available in PDF format on the disc.

Contents

Contents i

1 Introduction 3
1.1 Problem Statement . . . 4

2 Analysis 7
2.1 LIP . . . 8
2.2 SQL . . . 11
2.3 HaskellDB . . . 17
2.4 Hibernate Query Language . . . 19
2.5 Discussion . . . 20
2.6 Summary . . . 22

3 Design 23
3.1 Limitations . . . 23
3.2 Example Scenario . . . 25
3.3 Functionality . . . 27
3.4 Syntax . . . 34
3.5 Semantics . . . 36
3.6 Type System . . . 40
3.7 Mapping . . . 43
3.8 Summary . . . 45

4 Implementation 47
4.1 Compiler Structure . . . 47
4.2 Code Generation . . . 55
4.3 Summary . . . 59

5 Test 61
5.1 Code Testing . . . 62
5.2 Validation . . . 63

5.3 Summary 66

6 Related Work **69**
6.1 Language Integrated Query 69
6.2 NHibernate 70

7 Evaluation **73**
7.1 Validation Test 73
7.2 Technical Test 74
7.3 Analysed Technologies 75
7.4 Functionality 75
7.5 Limitations 78
7.6 Tools 80

8 Future Work **81**
8.1 Language Integration 81
8.2 Object-relational mapping 82
8.3 Industrial Application 83

9 Conclusion **85**

Bibliography **87**

A Grammar **93**

B Amigo manual **103**

C Validation test **113**

D Summary **119**

List of Figures **121**

Listings **122**

Acknowledgements

We would like to thank our supervisor Lone Leth Thomsen for her invaluable help and constructive criticism for this project.

We would also like to thank a few of our fellow students, namely Jacob Elkjaer Hansen, and Thomas Legaard Kjeldsen for agreeing to participate in the testing of Amigo, even though they had to work on their own projects.

Chapter 1

Introduction

In developing software today, a popular approach is to make use of an object oriented programming language to describe the logic and model of the application, and a relational database as data storage. However, as popular as this approach is, there are problems associated with it. The general problem concerning the interaction between the object oriented world and the relational world is often referred to as the impedance mismatch problem [20].

Traditionally, querying a database from an object oriented programming language involves the use of a call level interface (CLI) such as ODBC [6] or JDBC [14]. Structured Query Language (SQL) queries are expressed in text strings which are sent to the database. This is also sometimes referred to as explicit query execution. Expressing the queries in simple strings does not provide syntax or type checking of those queries. The result of this, is that errors in the queries are not caught until runtime, extending development time and causing frustration for the programmer. We will refer to the issue of explicit queries not being type checked at compile time as weakly typed explicit query execution.

We worked on solving this problem in the development of an extension to the C# language called Language Integrated Persistence (LIP), described in [16]. LIP tries to solve, at least to some extent, the impedance mismatch. By extending C# with a number of constructs and keywords LIP relieves the application programmer from being concerned with the database, and instead focus on the development of the application. The database is hidden from the programmer and the database automatically reflects the application. However, in the final design of LIP the database shined through. We realised that if one wants to use more than basic database function-

ality the database cannot be completely hidden from the programmer. In order to utilise the functionality provided by the Database Management System (DBMS), the programmer needs to be aware of the database, which in effect means that there is no reason for hiding it completely from him.

In this project we are concerned with creating a solution for expressing database queries that, on the one hand, is well suited for use in an object oriented language and, on the other hand, provides the use of DBMS specific functionality while also making sure that queries are correct.

The solution is implemented as a query language, called Amigo, which could later be integrated in an object oriented host language. Unlike LIP we do not try to hide the database from the programmer. We assume that most programmers have some database experience and will find it difficult to completely let go of the database. In other words, most programmers are comfortable thinking about data represented as tables, but have difficulty handling complex table operations. It is often when programmers have to create SQL queries where several tables are combined that problems occur. We want to alleviate the programmer from having to concentrate on this.

Although LIP was not the right solution some of the ideas from it are worth keeping. Most notably the idea of of using filters. Filters are method like constructs which can be used on groups of data and in the context of relational databases they represent SQL queries. LIP is presented in greater detail in section 2.1.

1.1 Problem Statement

In this project we address the problem of weakly typed explicit query execution performed in object oriented programming languages on data stored in relational database management systems (RDBMS). The primary goal is to extend the filter concept from LIP and develop it further to utilise more of the RDBMS's features. The new concept is to be implemented as a query language called Amigo. Because Amigo access data stored in relation databases it has to be compatible with SQL. The filters should comply with the object oriented model but the relational model should be clearly visible. The programmer should be aware of when he is using the database and when not. The specific goals of this project are to analyse known languages and technologies, that deal with similar issues, to identify which features are to be available in Amigo, design the language creating a complete syntax and informal structured semantics. The core features of the language

are to be implemented along with some additional features implemented as proof-of-concept. Unit tests are to be written for each feature implemented, and to test the usability of the language a test with a number of people is to be devised and performed.

Chapter 2

Analysis

Before designing and implementing a new query language, it is important to investigate existing solutions in order to gain a broad understanding of the work and experiences of others. We begin this chapter by describing the LIP project which is the basis of the Amigo project. Then we describe the query languages and technologies we have found of particular relevance to this project. They are:

HaskellDB: A query library for the Haskell programming language. HaskellDB is interesting because it has some filter-like language constructions.

HQL: The query language used by the object oriented persistence framework Hibernate. HQL is interesting because it is the query language from a popular object relational mapper.

PL/SQL: The language used in the Oracle Database. PL/SQL is interesting because it extends standard SQL with some interesting features.

T-SQL: The language used in MSSQL. T-SQL is interesting because it extends standard SQL with some interesting features.

Each of these languages and technologies have a different approach to the concept of data extraction from a database. This fact makes it easier to identify their strong and weak points in relation to each other and also to determine which features should be adapted by Amigo. Following the analysis of the existing solutions, the features and functionality of these are discussed, as well as how they apply to this particular project.

The analysis serves as both a general exploration of existing solutions but also as a way of determining how others have dealt with specific problem areas. In the introduction we presented a few basic requirements we have for the language. Amigo has to be SQL compatible and make use of a

method like construct called filters. The aim of the analysis is on one hand to investigate how, or if, others have found a solution to these issues and on the other hand if others have additional features which we have not thought of. We have expressed a desire to make use of the DBMS as much as possible, for instance to let the DBMS perform aggregate functions instead of the host language.

In the following we focus on retrieval of data. Several of the technologies also support insertion and update of data, but this is not relevant to this project.

2.1 LIP

This section introduces the project [16] that lead up to the Amigo project. The project aimed at finding a solution to the impedance mismatch problem. The solution was implemented as a language called Language Integrated Persistence(LIP), an extended version of C#. Unlike Amigo, LIP was designed to handle everything that had to do with connecting object oriented applications with relational databases. We decided on a very pure or clean approach in which the programmer did not see the database at all. Clean or pure refers to the fact that we chose one paradigm and not a mix of several paradigms. Or rather, this was the original plan. The idea was that the programmer would write the application with no regard to the database. The database would then be generated automatically at compile time. The choice of hiding the database meant that all data operations would have to be handled in the application. This realisation came to us some way into the development, and it made us reconsider the choices we had made. We reasoned that if you were to do all data operations in the application, many of the arguments for using a relational database would be gone. In many situations using an object oriented database would be much easier. If we wanted to create a language that would appeal to people who for some reason were forced, or preferred, to use rational databases, we would have to provide ways of utilising database functionality.

The project still had as a goal to hide much of the database, but it was important that the programmer could utilize database functionality. The basic design idea was a filter construction that should make it out for the `WHERE` clause and any sorting or limitations in a corresponding SQL query. The filter construction resembles a method and it is intended to be used just like a method. As the goal was to handle all database related areas,

data retrieval was not the only thing we had to address.

We decided, as we had already broken with the idea of hiding the database, that programmers would feel more comfortable being able to control what should be persisted and when it should be saved and deleted. The first part was accomplished by treating all objects as transient unless otherwise indicated. The programmer indicated which fields should be persisted by using the `persistent` keyword that we introduced. Although technically unnecessary, we decided that if there were persistent fields in a class the class declaration itself had to contain the `persistent` keyword. This was done as a help for the programmer, so that if he had a class that at some point during development was persistent, but later should not be, he could be sure that no fields where accidentally persisted. The second part was accomplished by introducing save and delete methods.

Before going on with our intentions with LIP here are two examples. Example 2.1 shows how the save and delete methods are used.

Listing 2.1: LIP example

```
1  Employee e = new Employee();
2  e.Name = "John Doe";
3  e.Save();                       // Object is inserted in database
4  e.IsManager = true;
5  e.Save();                       // Object is updated in database
6  e.Delete();                     // Object is deleted in database
```

Example 2.2 shows a persistent class and what a filter looks like and how it is used.

Listing 2.2: LIP example

```
1  public persistent class Employee {
2      public persistent primary int Id;
3      public persistent string Name;
4      public persistent int HourlyRate;
5  }
6
7  public persistent class WorkSession {
8      public persistent primary int Id;
9      public persistent Employee Employee;
10     public persistent int Hours;
11
12     private filter WorkSession ByEmployee(Employee e) {
13         Employee == e;
14     }
15 }
```

```
16
17 public class PaymentCalculator {
18     private WorkSession session;
19
20     public PaymentCalculator() {
21         session = new WorkSession();
22     }
23
24     public void CalculatePayment(Employee _emp) {
25         int TotalPayment = 0;
26
27         foreach (WorkSession s in session.ByEmployee(_emp)) {
28             int SessionPayment = s.Hours * _emp.HourlyRate;
29             Console.WriteLine("Session payment {0}", SessionPayment);
30             TotalPayment += SessionPayment;
31         }
32
33         Console.WriteLine("Total payment {0}", TotalPayment);
34     }
35 }
```

As mentioned, introducing indication of persistent fields and save and delete methods went directly against the language requirement of hiding the database. We did not see a way around this problem. As we were already leaning more and more towards the idea of showing parts of the database, we determined it to be the best solution. We reasoned that most programmers would have some knowledge or experience with databases. They would be comfortable with the concept of data represented in tables and the basic mapping between tables and objects. The real problem for most programmers is that database queries where several tables have to be combined quickly become complex and difficult to write.

In the development of LIP we went from, what was perhaps, a naive idea of a clean language to a more realistic idea of simply making the use of a database from an object oriented language as easy as possible. This is essentially the goal we have brought into the Amigo project. The problem with LIP was twofold. It ended up being a mix of several ideas and it made it somewhat confusing to use. It was obvious that many of the constructions were not part of the original idea. The second problem was that we could see features such as aggregate functions and union etc. would be very difficult to include without making LIP even more confusing to use. Developing LIP gave us some good ideas but we had to go back and start from scratch if all the ideas were to work.

2.2 SQL

The Structured Query Language which is specified by the SQL92 [17] and SQL99 [18] standards is the query language of choice in the most widely used RDBMS's. Most of the RDBMS's implement a superset of SQL, which includes extended functionality and/or containing language features known from existing programming languages. We have chosen to investigate the features found in two of these supersets of SQL. The two languages we will look at in the following is PL/SQL used in the Oracle Database and T-SQL used in the MSSQL Database. It is important to note that T-SQL and PL/SQL are supersets of SQL so both the extensions to SQL and the SQL language itself are discussed in this section.

2.2.1 Motivation for extending SQL

Using SQL, the programmer is limited to executing single or batches of declarative statements. In T-SQL and PL/SQL, features are introduced to give the ability to perform more advanced data focused tasks directly in the database using many known programming language constructs like iterative and conditional statements. Even though the languages are powerful, they are mainly used for administrative tasks concerning the database system and the data stored within as the flexibility of a general purpose language is missing. However, the languages can still be useful in advanced applications as the programs written in T-SQL and PL/SQL can be called in the same way that SQL can be used from general purpose programming languages.

2.2.2 Main features

Both T-SQL and PL/SQL programs are executed in an imperative fashion and have the common features mentioned here:

- Variable declaration and assignment
- Procedure declaration
- Conditional statements
- Iterative statements

Assignment to a variable can be the result of an SQL query. This variable can be used in iterative statements to iterate through relational data and perform operations on this data.

As we might host Amigo in a general purpose programming language, it is important to note the ability to use queries in this fashion. However, the

ability to declare procedures and use iterative and conditional statements is of little interest as these features are available in the host language.

The SQL92/99/2003 specifications describe the syntax of a `SELECT` statement and the basic functionality is the ability to retrieve zero or more columns from one or more tables with the possibility of using the basic SQL features listed here:

- Ordering rows
- Grouping rows
- Filtering rows
- Joining columns from multiple tables using key-relationships

All of these features are well documented, for instance in [19] and will not be discussed further in this chapter.

2.2.3 Key data extraction features

Set operators

The result of a SELECT statement is a set of tuples. Operations for working with multiple sets of tuples are available in both T-SQL and PL/SQL. The combined set operators in the two languages are listed here:

- `UNION`
- `UNION ALL`
- `INTERSECT`
- `MINUS`

They all work by combining two sets, but each with different semantics. `UNION` takes all distinct rows from either set and places them in the new set. `UNION ALL` takes all rows from the two sets and places them in the new set, including duplicates. `INTERSECT` takes all distinct rows present in both sets and places them in the new set. Lastly, `MINUS` takes all distinct rows, from the first set, not in the second set and places them in the new set.

Subselects

Using the result of one query execution in another is possible in T-SQL and PL/SQL by assigning the result of a query to a variable. A more powerful approach is subselects which gives the developer the ability to use the result of one query as a part of another for instance in the WHERE clause. In both

T-SQL and PL/SQL operators for working with lists are present. These can be used in the context of sub-queries, as in this T-SQL example:

```
SELECT Name, Age
    FROM Employees
    WHERE EmployeeID
    IN (SELECT EmployeeID
        FROM DepartmentEmployees
        WHERE Department = 'Sales')
```

The type of query presented in the example is very useful when querying data which is part of a many-to-many relation in the database as you can easily use the table, which is used for modeling the relation, in your `WHERE` clause using a subselect.

In addition to using the `IN` operator on a statement, it can be used on a set of numbers, hence retrieving for instance a list of rows given by their primary key:

```
SELECT Name, Age
    FROM Employees
    WHERE EmployeeID IN (1,20,40);
```

Other operators exist that can be used with subselects, for instance the `EXISTS` keyword which can used to check if a subselect returns a result. In the following example only Departments that has employees associated is returned:

```
SELECT * FROM Departments WHERE EXISTS(SELECT * FROM WorksIn
WHERE DepartmentID = Departments.DepartmentID);
```

The same query can easily be rewritten using `IN` instead:

```
SELECT * FROM Departments WHERE DepartmentID
IN(SELECT DepartmentID FROM WorksIn);
```

The `EXISTS` provides a shortcut to these type of queries that could be created using `IN` and just compare the primary key as seen in the example.

2.2.4 Built in functions

Both T-SQL and PL/SQL have a large number of built in functionality used for mathematical and various other tasks.

Functions that vary from what we know from most programming languages are those that operate on a set of single or multiple rows returned from statements. These statements are referred to as Single-Row functions and Aggregate functions.

Single-Row functions return a single result for each row resulting from a query. An example of a Single-Row function could be the square root function `SQRT`. A query on the form

```
SELECT SQRT(Salary) FROM Employees;
```

would return one row for each row present in `Employees` containing the square root of the value present in `Salary` of that row. We have divided Single-Row functions into categories which gives an indication of which data types they can be applied to:

- Date and time functions
- Mathematical functions
- String/Character functions
- Null handling functions
- Conversion and cast functions
- Comparison functions
- Text and Image functions

One special group of functions is the null handling functions. This has a special significance in database systems because of the fact that all data types can be `NULL` and that an operator applied to a value and a `NULL` value will always return `null`. The last group of functions called Text and Image functions is used to work with large amounts of data in either text or binary form stored in the database.

Aggregate functions operate in a completely different fashion. An aggregate function returns a single result row based on a set of rows. An

example of an aggregate function could be the `AVG` function which returns the average of a column in a set. A query on the form

```
SELECT AVG(Salary) FROM Employees;
```

would return a single row containing the average of all the values of `Salary` present in `Employees`. Many different aggregate functions are available. All share the characteristics of `AVG` in the fact that they return a single row result operation on a set of rows. Often aggregate functions are used with the `GROUP BY` clause to retrieve only the aggregate value for certain groups of rows present in a table. An example could be to find the average salary in several departments in a company's employee database:

```
SELECT AVG(Salary) FROM Employees GROUP BY department;
```

This would return one result row for each department containing the average salary for employees related to that specific department.

Interval/Ranges

In SQL's conditional clauses, the developer has the opportunity to work with ranges. That is, to specify a range of, for instance, integers that should be given for a certain column in the database. This is done using the `BETWEEN` keyword. For instance, the following query will return all rows from the `Employee` table where the value of the `Salary` column is between 10.000 and 20.000:

```
SELECT * FROM Employees WHERE Salary BETWEEN 10000 AND 20000;
```

Looking at the above purely from a language perspective, and not thinking of how the database can use this special syntax to optimize queries, it is equivalent to the following query using the comparison operators which is a part of SQL:

```
SELECT * FROM Employees WHERE Salary >= 10000 AND Salary <= 20000;
```

Hierarchical queries

PL/SQL has a feature for working with hierarchical data stored in a database with parent-child relationships between rows in tables. This is accomplished by adding extra functionality to the SELECT statement from SQL. The syntax for querying hierarchical data is shown here:

```
SELECT-STATEMENT START WITH condition
CONNECT BY [NOCYCLE] condition
```

The underlined words are keywords and the first condition must contain an expression prefixed with the PRIOR operator which indicates the reference to the parent row.

If this feature was not included, as is the case with standard SQL implementations, the hierarchical data could only be retrieved by

1. Using multiple queries(recursive database calls or recursion in TSQL using a temporary table as a stack
2. Let the table join itself, however its only suitable for getting paths and not getting all the columns
3. Maintaining a nodepath in a seperate column in the database and sorting by this to get the rows in their correct order
4. Using various techniques like nested sets or nested intervals which makes it very complex to insert new rows

An example of using a table joining itself i shown below:

```
SELECT Top.Name Top, Second.Name Second, Elements.Name Element
FROM Elements
INNER JOIN Elements AS Second ON Elements.ParentID=Second.ElementID
INNER JOIN Elements Top ON Seconsd.ParentID=Top.ElementID;
```

This will show all paths in the hierachi for instance if the hierarchy is

```
Element1
- Element1.1
-- Element1.1.2
--- Element1.1.2.1
--- Element1.1.2.2
```

The above query will return the following rows:

Element1	Element1.1	Element1.1.2
Element1.1	Element.1.1.2	Element1.1.2.1
Element1.1	Element.1.1.2	Element1.1.2.2

which is the path of the subnodes from the parent, however if we want all the data for each row the query becomes more complicated and one of the more advanced models would be better suited.

In PL/SQL it can easily be done using `START WITH...CONNECT` BY as shown here:

```
SELECT lpad(' ',level*2,' ')||Name
   FROM Elements
   START WITH ParentID IS NULL
   CONNECT BY PRIOR ElementID = ParentID
```

This will print the whole tree with nice padding and sorted, the padding can be left out and the level can be retrieve.

2.3 HaskellDB

Haskell[8] is a functional programming language. HaskellDB[4][3] is a library for Haskell that provides a means of communicating with a back-end DBMS from a Haskell program. Instead of using explicit queries in strings, HaskellDB enables the programmer to express those queries using standard Haskell functions. Using the Haskell type system, the queries are checked at compile time and thus a type safe way of expressing database queries are the result - in contrast to strings embedded explicit SQL queries.

Consider the following SQL query:

```
SELECT * FROM Employees WHERE fname = "John"
```

This simple query can easily be expressed using HaskellDB:

```
employees = do
    x <- table employee;
    restrict (x!e_fname .==. constant "John");
    project (e_fname = x!e_fname, e_lname = x!e_lname)
```

The table `employee` is bound to the variable `x`. The `restrict` function is equivalent to the `WHERE` clause in standard SQL. The `(!)` operator is used to select attributes on a table — or relation as they are called in HaskellDB terms. The `(.==.)` operator is the relational comparison operator for equality. All relational comparison operators available in SQL are also available in Haskell. The `project` function is used to create a projection of the `e_fname` and `e_lname` attributes. `employees` is of type `Query`, and it simply returns the SQL query equivalent to the HaskellDB query.

HaskellDB also provides the ability to use aggregate functions on relations. Consider this example:

```
countProgrammers = do
    x <- table employee;
    restrict (x!e_role .==. constant "Programmer");
    project (e_role = count x e_role)
```

The code in the example counts the number of employees who are programmers. As can be seen, the `count` function is given the variable that holds the relation, in this case employee, and the attribute on which the count should be performed. Although aggregate functions are often available natively in the DBMS, HaskellDB implements these functions as part of the library.

Ordering works in a similar manner as the count aggregate function described above. The function `order` takes a list of attributes, and orders according to that list either ascending or descending.

2.4 Hibernate Query Language

Hibernate is an Object Relational Mapper(O/RM) API for Java. The Hibernate Query Language (HQL) is Hibernate's query language designed to look like SQL but with object orientation in mind. HQL is similar to SQL as it uses many of the same keywords, and queries may consist of clauses, aggregate functions, and sub-queries.

Let us start with a simple example:

```
SELECT * FROM Employee AS employee
```

Writing `FROM Employee AS employee` will return all instances of object `Employee`. The alias `employee` can be referenced in the rest of the query. HQL supports polymorphism, thus writing the above query also returns all subclasses of `Employee`.

It is possible to specify more precisely what is to be extracted. Writing `SELECT employee.name FROM Employee AS employee` will return all values of `name` in all instances of `employee`.

In the `WHERE` condition all the standard logical operators known from SQL can be used. `ORDER BY` and `GROUP BY` also work as they do in SQL. Furthermore, inner, left outer, right outer, and full joins are available and working as in SQL. The following aggregate functions are available:

- avg()
- sum()
- min()
- max()
- count(*)
- count(...)
- count(distinct ...)
- count(all...)

SQL arithmetic operators, concatenation, and recognised SQL functions are allowed in the `SELECT` clause. As mentioned, subselects are supported by HQL but only if the underlying database supports it. Subselects work just like in SQL, or more precisely, they work like other HQL queries. HQL queries return results as objects or arrays of objects. Hibernate has some helper functions which allow the results to be iterated easily. Also it is possible to force the returned results to be typecast. These features are essentially not part of HQL, but a part of Hibernate, so we will not delve deeper into the effects of these features.

Let us look at a larger example [5]:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc @@
```

This query returns the order id, number of items and total value of the order for all unpaid orders for a particular customer and given minimum total value, ordering the results by total value.

2.5 Discussion

As stated in our problem statement, one of the goals of this project is to design a query language suitable for hosting in a general purpose object oriented language. Because of this fact some of the features described in the previous sections will be implemented using constructs similar to those of the host language. In this section we discuss which features are relevant for use in the scope of the Amigo query language.

2.5.1 Tables or Objects

We design Amigo to operate on objects like seen in HQL and not on tables like SQL and HaskellDB. The mapping between objects and tables is fairly straightforward so the programmer can easily identify the tables and columns even though he is dealing with objects. The mapping will not be discussed in much detail, as well known solutions to this problem have already been suggested by various O/R-Mappers like Hibernate. However, the features of the database are to be present so we need to specify a syntax that allows for the features normally applied to tables to be applied to objects.

2.5.2 Combination of filters

In Section 2.1 we described the idea of a filter and the fact that it is a method-like construct in the general purpose programming language, and correspond to a single SQL statement. It is an obvious choice to allow the user to apply functions from SQL, which are normally applied to one or more queries, to one or more filters. This means that we have to have operators that work on two filters to perform tasks like `UNION`, `JOIN`, and `INTERSECT`. And while these operators are applied to two filters we allow the code generation to combine those to a single query to be executed by the database system.

Another possibility for combining filters is the idea of subselects. In our case we have chosen to abandon the idea of writing one "anonymous" query inside another to keep our syntax simple. Instead we give the user the possibility to use one filter as part of another filter's body through a kind of call. Access modifiers should be in place for filters just as with methods, so filters that are only used as part of other queries is not accessible from outside the class where the filter is present.

2.5.3 Built in function

The built in functions available in T-SQL and PL/SQL will have to be available in the filter bodies. These functions are often used and considered by many to be essential when using SQL. The functions should work just like they do in T- SQL and PL/SQL because they are so well known. Functions such as `SQRT` and `AVG` can be directly mapped to Amigo. Functions such as the range function using the `BETWEEN` keyword is not as easily mapped.

2.5.4 Query mechanisms

All of the query languages we have investigated in this chapter provides the ability to specify constraints on queries. In the SQL based languages, the `WHERE` clause is used to specify the constraints. In HaskellDB, the `restrict` function is used. It is clear that Amigo must support this ability as well. Without it, the flexibility and expressiveness of Amigo filters is greatly reduced - and the language will be largely useless.

Along with being able to specify constraints for queries, it is also necessary to be able to sort and group the results of a query. The functionality of the standard SQL features `ORDER` and `GROUP BY` is also available in Amigo.

2.5.5 General purpose features

As we have seen in both PL/SQL, T-SQL, and HaskellDB, it is possible to use some general purpose functionality in the query languages. This is also possible in Amigo but these features are used just as the ones in the host language. That is, a conditional statement in the filter will be on the same form as the one in the host language, and the code generated will still only be SQL not using these conditional statements. These features makes it easier to write general purpose filters.

Variable assignments are also possible in both PL/SQL and T-SQL and this notion can be very useful to avoid multiple queries to the database. A variable assignment, with results of named filters, in the body of a filter is translated to a variable assignment in the generated code, hence sending only one query to the database. The results of all variables are returned by the query so that the contents of variables after query execution is as expected by a normal assignment done elsewhere in the host language.

2.5.6 Advanced features

Some advanced features could help the programmer write far less code by hand. The functionality in PL/SQL to work with hierarchical data is one of these features. We introduce an operator for this, and then generate recursive joins to the level needed to return a tree-structure.

2.6 Summary

In this chapter we have analysed a number of existing query languages and technologies, in order to gain a broader understanding of the features of existing solutions that might be of relevance to us in this particular project. The analysed technologies were Hibernate HQL, HaskellDB, PL/SQL, and T-SQL. Each of these have features that are specific to them individually, but most are seen in all of the technologies. Along with this analysis, the LIP programming language was described in order to communicate the background and rationale for the Amigo query language. The features of LIP and the analysed technologies was the basis for a discussion on which features could be adapted into Amigo. Important results of this discussion includes the possibility of combining filters, and the ability to use aggregate functions within the filters.

CHAPTER 3

Design

During the analysis of the already existing query languages for database systems, we have learned what features are crucial for the success of a query language. In this section the syntax for utilising these features in our query language is specified as well as the semantics of the constructs introduced in the syntax. First, however, we establish a number of limitations for Amigo.

3.1 Limitations

Designing and implementing a programming language is a large and complex task. There are many issues that must be addressed and discussed in order to make sure that the language meets its goals. Once the language is adopted by programmers it is very difficult to make changes to the language. Furthermore, it is often the norm that it takes several years before a new programming language breaks through into mainstream. Because of these facts, we have found that it is necessary to introduce some limitations to the functionality and scope of the Amigo project. This section briefly describes these limitations.

3.1.1 Queries

Due to the fact that the Amigo language is a direct derivation of the LIP filter concept, an obvious limitation of this project is that Amigo only concentrates on selection queries. Amigo - in its current form - does not provide any functionality to insert, update, delete, or alter data or the database itself. We find, that the most interesting aspect of query languages is the actual selection of data. Insertion, updating, and deletion are relatively

trivial features of a query language, but is by the LIP filter definition not a part of the filter concept and thus not applicable in the focus area of the Amigo query language.

3.1.2 Mapping

In order to provide static type-checking for a database query language, it is necessary that the type checker is aware of the actual database schema. The database schema has information about tables and columns. The type checker compares references to elements in the database schema from within the language with the database schema itself, to make sure that compatible types are used on both sides. Ideally the static type-checking is done by means of direct communication between the compiler and the database. In this project we have decided to rely on Hibernate mapping files to represent the database schema. Hibernate mapping files provide a simple means of representing the schema using XML, which is easily parsed into a data structure that can be used in the compiler. This subject is more thoroughly discussed in Section 3.7.

3.1.3 Language Integration

As was mentioned in Chapter 1, integrating Amigo in an existing programming language, for instance C#, is an issue that is not in the focus area of this particular project. It is, however, a distinct possibility that it will be done at a later time. To that end, we are concerned with designing Amigo such that the language is ready for future integration. Integrating a language like Amigo in an object oriented host language should provide a low coupling between the two languages, which would require an extensive analysis of how Amigo could interface seamlessly between the object oriented language and the relational database without having to compromise with either of the two. Normally, an object oriented model of the relational database schema is needed to make sure that the data retrieved from the database can be correctly represented in the host language. This kind of object/relational mapping is outside the scope of this project, and we will not get into the integration of Amigo in a host language.

3.1.4 Object orientation

In the previous subsection it was mentioned, that Amigo is likely to be integrated in an object oriented host language some time in the future. This

means that, unlike SQL, Amigo takes a more object oriented approach to the way queries are expressed and represented. However, object orientation is a relatively complex subject, that involves many aspects. In this project we will disregard issues like inheritance and polymorphism, in order to keep focus on creating a clean and easy-to-use query language. These features should be added in future versions of the Amigo query language.

3.1.5 Testing

Testing a new programming language is important to make sure that all semantics work as intended. Although we do plan to test Amigo in the this project, it is not possible to carry out an extensive user test with several programmers, due to the simple fact that we have to both design and implement the language and compiler in the course of the project period. This leaves little time to perform user testing. Therefore, we will limit the user testing to 2-3 programmers in this project. This gives us an idea of how programmers will receive the Amigo language, but we will not catch all problems associated with the language in its initial version.

3.1.6 Performance

In order for a programming language to become successful it is important, among other things, that there is an efficient compiler or interpreter available for the language. Measuring performance of compilers and interpreters requires considerable effort to cover all aspects of the application of these tools. Since we, in this project, mainly are concerned with designing the Amigo language and implementing a compiler, that shows the capabilities of the language, we will not be focusing on the performance of the compiler — let alone possible performance enhancements.

3.2 Example Scenario

In order to make sure that as few misunderstandings as possible occur while reading this chapter, we briefly describe a scenario that will be used for all of the Amigo code examples throughout the rest of the report. The example scenario is described using an ER-diagram of a database for a fictional company. Figure 3.1 shows the diagram.

As the diagram shows, the database consists of a number of tables `Employees`, `Departments`, `Products`, `OrderLines`, and `Customers`. Each

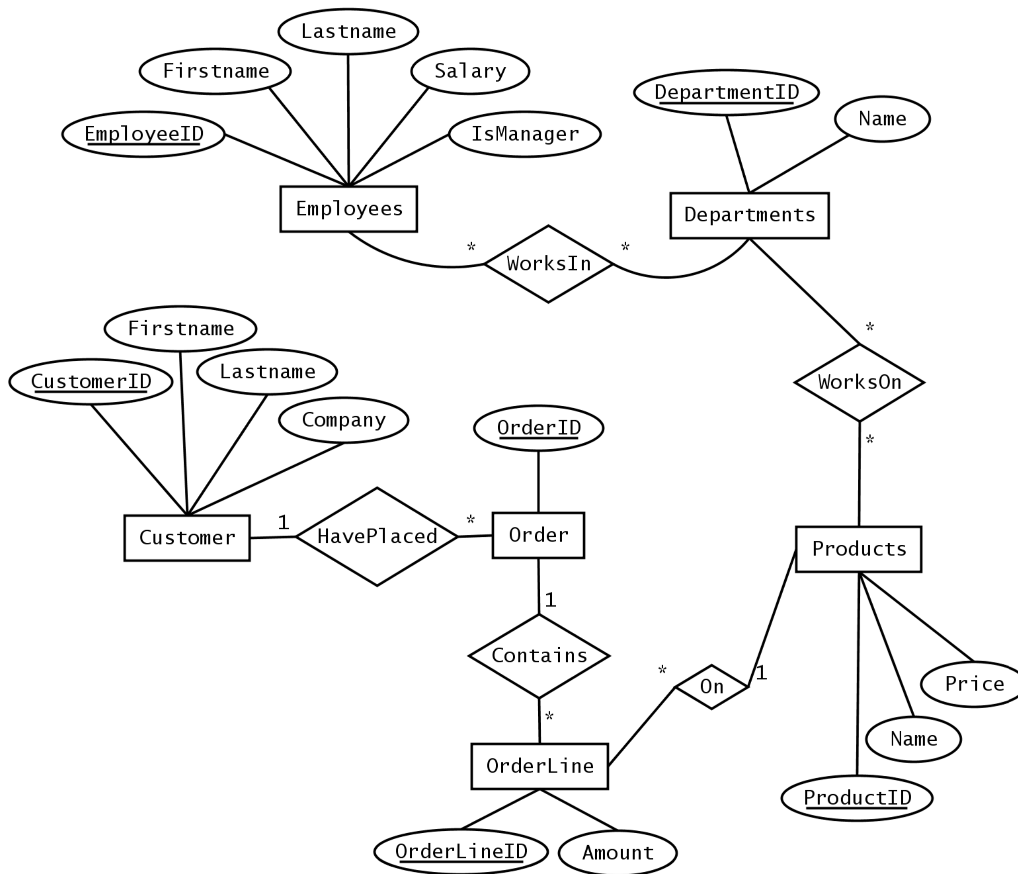


Figure 3.1: Diagram of example scenario

table has a column that is set as the primary key, along with a number of other columns. Between the tables there are relations `WorksIn`, `WorksOn`, `On`, `Contains`, and `HavePlaced` - each with associated cardinalities. Note, that tables in the database are named in plural. When referring to these tables in Amigo, they are named in singular. This is due to popular naming conventions in both the relational and object oriented world.

As mentioned, this example scenario is used as a basis for the examples that describe the Amigo language and its features in the rest of the report. Along with this, it is also used for testing the language. This is more thoroughly described in Chapter 7.

3.3 Functionality

The feature-set available in Amigo is partly derived from the query languages described in the analysis and partly from our own experience working with both relational data and object oriented programming languages.

The main structure in Amigo is the filter construction, known from the LIP programming language, which corresponds to an SQL query. Each filter is to be named and hosted in a method-like construct that could, at some point, be integrated in an object oriented programming language. Furthermore filters can be used within filters to achieve the power of SQL's subselects. New filters cannot be defined inside a filter but a previously defined filter can be used.

Amigo is not meant to be a complete general purpose programming language. This means that it is going to be used in connection with another general purpose language. It is not clear how this is going to be devised but one possibility is to implement a pre-processor. In any case the filters must be self contained. It has to be possible to evaluate filters without knowing the context. This makes sense in more than one way. First of all it makes reading the code easier and second of all it is a way of keeping all options open since we do not tie Amigo to one general purpose language or one way of using it.

In Amigo, a filter declaration looks very similar to a method declaration in object oriented languages such as Java or C#. If Amigo at some point is integrated in an existing language, the filters will not look like complete aliens in the host language's syntax, but rather resemble the already available constructs. A filter declaration looks like this:

```
public var NameOfFilter(int arg) {
}
```

The reason for using the keyword `var` here, is that in this project we are not very concerned with the return types of a filter, since this is mainly a matter where a complete object/relational mapping is present. `var` is simply a placeholder for the return type. This issue is more thoroughly discussed in Section 3.6

Moving on to the internals of filters we continue to follow the object model and create blocks for each part of the filter. The first block which is called the `object` block is used to state what kind of objects to return. If

we look at the example scenario presented in Section 3.2 a filter that would return a collection of employees is written like this:

```
public var Employees() {
    object:
        Employee;
}
```

The filter header states that the filter is public, the type is inferred, and the name of the filter is `Employees`. The filter contains a single block, namely an `object` block which in turn contains a reference to the `Employee` object. Converting this filter to SQL would result in:

```
SELECT * FROM Employees
```

The query returns all employees from the `Employees` table.

We can now retrieve all objects of a given type. The next issue is to impose restrictions so that only objects which have certain attributes are returned. In SQL this is handled in the `WHERE` clause. We borrow this idea and introduce a new block called the `where`-block. We keep the name because programmers are used to the terminology and will inevitably think in terms of SQL when they write Amigo code. There is no reason to change the terminology as it will most likely just lead to confusion. For the `where`-block to have any effect we introduce boolean expressions. Boolean expressions can be constructed using comparison operators to get boolean values from comparing values and variables. The comparison operators supported in Amigo are:

- !=
- ==
- <=
- >=
- <
- >

It is also possible to use **true** and **false** values.

The `where`-block can contain any number of boolean expression. Each expression is ended with a semicolon. To make it easier to write the `where`-block the default behavior is to assume that the boolean expression have

an **and** operator between them. So if we want to get all employees with salaries above 10.000 and who are also managers we simply write:

```
public var HighPaidManagers() {
    object:
        Employee;
    where:
        Employee.Salary > 10000;
        Employee.IsManager == true;
}
```

On the other hand if we want to get all employees with salaries above 10.000 or who are also managers we write:

```
public var HighPaidOrManagers() {
    object:
        Employee;
    where:
        or {
            Employee.Salary > 10000;
            Employee.IsManager == true;
        }
}
```

It is legal to write `and` in the first example but it is not necessary.

Having only the `object` and `where` block, filters can only return objects or collection of objects. We introduce the `value` block which allows the programmer to state that the filter should return something other than and object or collection of objects, for instance a single value or collection of values.

One of the problems we had when developing LIP was that we wanted to make use of the database's aggregate functions. Keeping with the idea of letting the programmer see the database aggregate functions are written just like they would be in SQL. Aggregate functions are allowed in the `where`-block and in the `value`-block. If it used in the `value`-block, Amigo infers that the returned result is a single value. For instance, if we wanted

to know the average salary of all the employees by using the `AVG()` function, supported by most databases, we would write:

```
public var AverageSalary() {
    object:
        Employee;
    value:
        AVG(Employee.Salary);
}
```

The `value`-block overrides the `object`-block and the filter return a single value and not objects of type `Employee`. In the `value` block we can see that the aggregate function is written like you would write in SQL. The argument to the function is the `Salary` property on the `Employee` object.

In Amigo, it is possible to combine filters in order to provide the same functionality as the SQL subselects. This is accomplished, not by declaring a filter within another filter, but by calling a filter in the same way as a method is called in a general purpose programming language.

Variables can be assigned values as you would expect, but they can also be used to hold results from filter calls. All of this requires a new block which we call `init`. Variables declared in the `init` block can be used in all other block in the filter. Being able to call one filter from another makes filters very modular and helps minimise the size of filters. Assume that we have a filter called `AverageDepartmentSalary` which takes a department id as argument and returns the average salary for that department. Using this filter we can now write a new filter:

```
public var LessThanDepartmentAverageSalary(int departmentId) {
    init:
        float X = AverageDepartmentSalary(departmentId);
    object:
        Employee;
    where:
        Employee.DepartmentID == departmentId;
        Employee.Salary < X;
}
```

This new filter called LessThanDepartmentAverageSalary also takes a department id as argument. In the init block we call the AverageDepartmentSalary filter with the department id. The result is assigned to variable X. The object block states that the return type is Employee. In the where block we state that we want all employees in the department in question and that they must have a salary below the average. In the latter we use X which holds the result from the AverageDepartmentSalary filter.

From our study of SQL, it is clear that an additional three blocks are required to make Amigo filters expressive enough to be of use. These blocks are join, order, and group. In the join block the programmer can join object as he would join tables in SQL and the supported types of join are the same. In addition we saw an opportunity to help the programmer by introducing the tree join which performs a recursive join. A join is written using an infix operator and the left/right side of the operator can be primary-foreign key relationship or any other to fields. The following example for retrieving all employees who work in a department with a given department name shows how a join is used within a filter:

```
public var EmployeesByDepartmentName(string name) {
    object:
        Employee;
    join:
        Employee.DepartmentID === Department.DepartmentID;
    where:
        Department.Name == name;
}
```

In the order-block, the programmer can specify that the result should be ordered by a specific property in ascending or descending order. To indicate whether it should be in ascending or descending order we use a prefix notation. We borrow the naming from SQL giving us [ASC] for ascending and [DESC] for descending ordering. If we want all employees in a department ordered by their salary beginning with highest paid, it is a simple matter of writing:

```
public var OrderedEmployees(int departmentId) {
    object:
        Employee;
    order:
        [DESC]Employee.Salary;
}
```

In the group-block, the programmer is able to use the database function for grouping results by a specific field. This is normally used in conjunction with aggregate functions. For instance if we want the average salary for employees in each department we can write:

```
public var AverageSalaryPerDepartment() {
    object:
        Employee;
    value:
        AVG(Employee.Salary);
    group:
        Employee.DepartmentID;
}
```

In Amigo filters, it is allowed to use `if-else` and `for` statements within the various filter blocks. These statements are mainly included in Amigo in order to provide the ability to write more generic filters. That is, filters that can be used in more than a single context. The example below shows how an `if-else` statement can be used within a filter.

```
public var (int modifier) {
    object:
        Employee;
    where:
        if(modifier == 1) {
            Employee.Salary <= 10000;
        } else if (modifier == 2) {
            Employee.Salary <= 10000;
            Employee.IsManager == true;
```

```
        } else {
            Employee.Salary <= 10000;
            Employee.IsManager == false;
        }
}
```

Amigo also provides the ability to use arrays within the filters. Arrays are used in exactly the same manner as in most programming languages. However, in Amigo they can be used in the `where`-block to specify that the value of a row in a certain column must be one of the values in the array. To clarify what we mean by this, the following example shows this functionality:

```
public var EmployeesSalaryArray() {
    init:
        int[] salaries = {5000, 10000, 15000, 20000};
    object:
        Employee;
    where:
        Employee.Salary == salaries;
}
```

The last feature is not essential but it is very useful in many circumstances. Limiting the number of returned object or values is applied in the `object` block as a prefix. There are three limitations:

- $[x..y]$ Limits from index x to index y
- $[..y]$ Limits to index y
- $[x..]$ Limits from index x

If we want only the first 20 employees out of all of them we write:

```
public var FirstTwentyEmplyees() {
    object:
        [..20]Employee;
}
```

3.4 Syntax

In this section, we briefly cover the actual syntax of the Amigo query language. This is done in order to give a full overview of how the language looks, and what is allowed in the bodies of Amigo filters, syntax wise. The grammar for the language is shown in Extended Backus Naur Form (EBNF). The grammar should be relatively easy to follow - especially when held against the code examples in the previous section, as well as the description of the language semantics in the following section. We will not get into the details of each production of the grammar for the same reason. We will, however, briefly explain the syntax for this particular EBNF grammar, since this tends to vary in some publications.

The name of a production always starts with a capital letter. Tokens are written inside apostrophes. Production names are placed on the left hand side of a `::=` in the production declaration. The body of the production are on the right side of that symbol. In order to group symbols or alternatives in the production `( .. )` is used. Separation is marked using `|`. Repetitions are marked using either `*` for zero or more repetitions, or `+` for one or more repetitions. Optional productions or symbols are marked using `?`. Furthermore, it is important to note, that some simple productions have been left out for space considerations. For instance, the `Number` production is not included here, but the meaning of it should already be clear; namely a number consisting of one or more digits between 0 and 9.

Listing 3.1 shows the grammar for the Amigo query language.

Listing 3.1: Amigo EBNF grammar

```
1  Amigo ::= (Filter)*
2
3  Filter ::= Modifier 'var' Identifier '(' (ParamList)? ')' '{' (Block)* '}'
4
5  ParamList ::= Type Identifier (',' Type Variable)*
6
7  FilterCall ::= Identifier '(' (ArgList)? ')'
8
9  AggregateCall ::= Identifier '(' Variable ')'
10
11 ArgList ::= Expression (',' Expression)*
12
13 Type ::= (SimpleType | Identifier) ('[' ']')?
14
15 SimpleType ::= 'int' | 'float' | 'string' | 'boolean'
16
17 Variable ::= Identifier ('.' Identifier)* ('[' Expression ']')?
18
```

```
19 VarDeclaration ::= Type Variable ('=' (Expression |
20                     '{' Expression (',' Expression)* '}'))?
21
22 Assignment ::= Variable '=' Expression
23
24 Limit ::= Number? '.' '.' Number?
25
26 ForStatement ::= 'for' '(' VarDeclaration ';' Expression ';' Assignment ')'
27                  MultiOrSingleStatement
28
29 IfElseStatement ::= 'if' '(' Expression ')' MultiOrSingleStatement
30                     ('else' 'if' '(' Expression ')' MultiOrSingleStatement*
31                     ('else' MultiOrSingleStatement)?
32
33 OrStatement ::= 'or' MultiOrSingleStatement
34
35 AndStatement ::= 'and' MultiOrSingleStatement
36
37 MultiOrSingleStatement ::= ('{' Statement* '}' | Statement)
38
39 Statement ::= ForStatement
40             | IfElseStatement
41             | OrStatement
42             | AndStatement
43             | OrderExpression
44             | VarDeclaration ';'
45             | Assignment ';'
46             | Expression ';'
47
48 OrderExpression ::= ('[' ('ASC' | 'DESC') ']') Variable ';'
49
50 Block ::= InitBlock
51         | ObjectBlock
52         | ValueBlock
53         | JoinBlock
54         | WhereBlock
55         | GroupBlock
56         | OrderBlock
57
58 InitBlock ::= 'init' ':' Statement*
59
60 ObjectBlock ::= 'object' ':' ObjectBlockBody
61
62 ObjectBlockBody ::= ('[' Limit ']')? Variable ';'
63
64 ValueBlock ::= 'value' ':' Statement*
65
66 JoinBlock ::= 'join' ':' Statement*
67
68 WhereBlock ::= 'where' ':' Statement*
69
70 GroupBlock ::= 'group' ':' Statement*
71
72 OrderBlock ::= 'order' ':' Statement*
73
74 Expression ::= RelExpression (JoinOperator RelExpression)?
75
76 RelExpression ::= SimpleExpression (RelOperator SimpleExpression)?
77
```

```
78  SimpleExpression ::= Term (AddOperator Term)*
79
80  Term ::= Factor (MulOperator Factor)*
81
82  Factor ::= '(' Expression ')'
83           | FilterCall
84           | AggregateCall
85           | Variable
86           | Number
87           | String
88           | Char
89           | Null
90           | True
91           | False
92
93  JoinOperator ::= '===' | '|=' | '=|' | '|~' | '~|' | '><' | '->'
94
95  RelOperator ::= '==' | '<' | '>' | '<=' | '>='
96
97  AddOperator ::= '+' | '-'
98
99  MulOperator ::= '*' | '/'
100
101 Modifier ::= 'private' | 'public'
102
103 SimpleType ::= 'int' | 'float' | 'string' | 'char' | 'bool'
104
105 True ::= 'true'
106
107 False ::= 'false
```

3.5 Semantics

In this section we will outline the semantics of Amigo in an informal manor. As seen in the syntax, in Section 3.4, the body of a filter consists of multiple similarly labeled structures called *filter-blocks*. Each of these blocks and the statements that are allowed within them are described.

3.5.1 General structures

Most of the language statements of Amigo are not specific to single filter-blocks, but can be in all of the available blocks. These include declaration of variables, `for` statements, and `if-else` statements. This is functionality known from most general purpose programming languages. The semantics of this functionality aims to mirror the semantics of that in the host language.

Declaration of variables are allowed in all blocks but it should be noted that semantics for variable declarations is different in the `init` filter-block. A variable declaration allocates space for an identifier with the given type in the current scope and can optionally assign a value to it at the same time. A collection of Amigo filters has a top scope. Each filter in the collection has its own scope. Furthermore, new scopes are created in the bodies of a `for` or `if-else` statement.

Along with the simple values that can be assigned to a variable, Amigo also supports the use of one-dimensional arrays. The basic semantics of arrays are very similar to those known from various general purpose programming languages. An array is a data structure with a fixed number of elements of the same type. In Amigo, an array can only be declared with its initial values explicitly stated. Elements of the array are accessed through the `[`*index*`]` notation, where *index* is an integer value that indicates the element's position in the array.

Arithmetic expressions are limited, as the implementation of these is not the focus of this project. Addition, subtraction, multiplication and division are supported through the use of binary infix operators $+$, $-$, $*$, and $/$. Operators can be used on both floating point and integer values. The precedence of the arithmetic operators is (evaluated in listed order, starting from the top):

- Multiplication
- Division
- Addition
- Subtraction

Boolean expressions are available and can be constructed using comparison operators to get boolean values from comparing values and variables. Furthermore the **true** and **false** values can be used.

Boolean values are also returned when using the built in comparison operators:

- $!=$ - Not equal
- $==$ - Equal
- $<=$ - Less than or equal
- $>=$ - More than or equal
- $<$ - Less than
- $>$ - More than

These operators have the same semantics as the corresponding operators in the host language.

In the `init`, `value`, `where`, and `order` blocks we allow the conditional statement `if-else` where the else part is optional. Several `if-else` blocks

can be nested to allow for multiple checks of conditions and optionally an `else` block at the end which will be executed if none of the conditions evaluate to true.

```
if ( CONDITION1 ) {
    STATEMENTS1
} else if ( CONDITION2 ) {
    STATEMENTS2
} else {
    STATEMENTS3
}
```

The semantics of the conditional statement above should be known to the user from general purpose languages. Also, in the `where` and `init` block we allow the iterative `for` statement known from imperative general purpose languages like C, C#, Java etc.

The `for` statement has three sections apart from the body that contains the statements of the iteration:

for-initializer where one or more variables can be declared. This is only executed once before the first iteration of the loop.

for-condition that must evaluate to true for the body of the statement to be executed. The condition is tested before each iteration.

for-iterator that can update variables in scope of the filter-block or variables declared in the *for-initializer*. This is executed after each iteration.

3.5.2 The init block

In the `init`-block the user can define variables containing the result from other filters. The advantage of declaring these variables in the `init`-block is that multiple database connections will not be used as the variable declaration will be translated to SQL code using either subselects, temporary tables, or variable declaration depending on the database used. The `init`-block can contain both `if-else` statements and for statements.

3.5.3 The object block

The `object`-block defines what kind of objects to return from the query unless a `value` block is present. If the return type is a single object or a strongly typed collection of the objects, the type is inferred using the `where`-block. If a condition is added in the `where`-block comparing a single value to a primary or unique column in the database the result is a single value otherwise the result is a collection.

3.5.4 The value block

The `value` block gives the developer the power to return results of queries not necessarily returning an object or collection of objects. This is useful for returning results of aggregate functions, single values, or collections of values. Function calls made in this block are attempted to be mapped to a function in the database. If this is not possible an error occurs.

3.5.5 The join block

The `join`-block is used to reflect the join statements allowed in SQL. However, multiple operators are introduced to join two tables. The join tables all reflect the corresponding join types in SQL and hence have the same semantics. In addition to the join types present in SQL we introduce the tree-join operator which performs a recursive join on tables representing a tree structure in the database. This operator will have the column that points to the parent on the right side and the column that points to the child column on the right and will retrieve all nodes in a branch according to conditions set in the where block.

The left/right side of the join operators (except tree join) can be a primary-foreign key relationship or any other two fields the user might want to join. Furthermore, we allow just for two object names to be on both sides and their primary-foreign key relationship in the database will be inferred from the mapping files if possible.

3.5.6 The where block

The ability to filter the data returned is done using boolean expressions working on properties on objects. They can be compared with simple types or values defined in the `init`-block or parameters send to the filter. The operators available correspond to the operators used in SQL's `WHERE` clause

of the `SELECT` statement. Variables declared in the `init`-block as results of other filters will be used as subselects, temporary tables or variable declaration if the database SQL implementation supports it.

If an array-variable is written on the right hand side of the boolean expression, the array is treated as a list of values, and the left hand side of the expression is compared to each of the elements in the list.

3.5.7 The group block

Results can be grouped just like in SQL. This gives the user the ability to group by a specific property or even to return an aggregate value based on grouping.

3.5.8 The order block

The `order`-block is used to sort the returned data either ascending or descending according to one or more properties. This semantics is equivalent to the `ORDER` statement when used in SQL.

3.6 Type System

The type system is an important component of a programming language. Depending on the design of the language, the type system can be more or less complex and comprehensive. Amigo is a statically strongly typed language. This means that all values and expressions are type checked at compile time, and in the source code variables must be declared of a specific type. This provides Amigo queries with a level of safety that is not present in traditional string based SQL queries. In this section the type system of the Amigo language is described.

3.6.1 Basic types

Amigo provides a small set of basic types. These types are well known from other programming languages, and we will not get into them in much detail. The basic types are:

- `int` - *Integer value*
- `float` - *Floating point value*
- `bool` - *Boolean value*

- `char` - *Character value*
- `string` - *String value*

These types are normally used when declaring variables, for instance in the `init`-block. Each type is rather self explanatory, but for complete clarification consider the following example:

```
public var TypeExample() {
  init:
    int a      = 1;
    float b    = 1.5;
    boolean c  = true;
    char d     = 'c';
    string e   = "abcd";
    ...
}
```

The above variables can be used in all blocks of the filter, since variables declared in the `init`-block are in the scope of the entire filter. However, because of the strong typing of variables, assignments to a declared variable must be of the same type as that variable. The following example assignments will result in type errors at compile-time:

```
...
a = 'c';
c = "false";
e = 10;
...
```

3.6.2 Type inference

Amigo allows for filters to be called from within a filter. This means, that the return value of a filter can be used in an expression. It also means, however, that in order to be able to type check the expression, the type of the called filter must be known by the type checker. Filters, in the current incarnation of Amigo at least, does not return data of a specific type. This is mainly due to the fact, that no actual mapping of the database schema

in the object model exists. In future versions of Amigo, the type inference of filters may be replaced by strongly typed filters. Consider the following example:

```
public var GetEmployee(int e_id) {
  object:
    Employee;
  where:
    Employee.EmployeeID == e_id;
}
```

The filter in the example will retrieve employees with the id specified by the `e_id` parameter. But how is the filter to know what to return in this case? Should it return a collection of employees or just a single employee? The Amigo type checker will infer, that the filter returns just a single employee, by looking up the *Employee* table in the database schema and conclude that the *EmployeeID* column of that table is the primary key, and thus unique for each record in the table. However, if the column was non-unique the return type would be inferred to be a collection of employees. Consider now the next example:

```
public var IsManager(int e_id) {
  object:
    Employee;
  value:
    Employee.IsManager;
  where:
    Employee.EmployeeID == e_id;
}
```

In the filter above, we again focus on the *Employee* table. However, this time we just want to know whether an employee is a manager or not. The *IsManager* column on the *Employee* table is used to determine this. But the filter still needs to know what to return. This is done by the type checker looking up the type of the *IsManager* column in the database schema. In this case it is a boolean, and since the `value`-block only can return a single value, the filter will return a single boolean value.

As mentioned, the `value`-block is used to return single values from filters. Therefore, it is particularly useful for returning the value of an aggregate function. The following example shows a filter that returns the average salary of all employees.

```
public var AverageSalary() {
  object:
    Employee;
  value:
    AVG(Employee.Salary);
}
```

The Amigo type checker will infer the type of the return value for the filter by determining the type of the value returned by the aggregate function.

3.7 Mapping

Hibernate, or more precisely the .NET version NHibernate, is a complete object/relational mapper. It has a query language, methods for saving and updating etc., and it handles data conversion. It does all of this based on information stored in XML files detailing relations between tables and classes. We refer to these files as mapping files. It is these files that we are interested in. As explained in Section 3.1 we opted for not type checking against the database. Instead we use NHibernate mapping files. An alternative was to devise a new XML structure similar to NHibernate's, but it is not clear what solution is optimal. It is not important for this project that the mapping files contain only the necessary information as the focus is on queries.

None of NHibernate's other features are used, in fact NHibernate itself is not used. The mapping files are independent of NHibernate and can be generated by hand or by one of several mapping file generators.

As we are not concerned with how a class is persisted, the examples in this section uses NHibernate's approach. A persistent `Employee` class could look like the one in Listing 3.2.

Listing 3.2: NHibernate

```
1  using System;
2
3  namespace Firm
4  {
5      public class Employee
6      {
7          private int employeeid;
8          private string firstname;
9          private string lastname;
10         private float salary;
11
12         public Employee() {
13         }
14
15         public int EmployeeID {
16             get { return employeeid; }
17             set { employeeid = value; }
18         }
19
20         public string Firstname {
21             get { return firstname; }
22             set { firstname = value; }
23         }
24
25         public string Lastname {
26             get { return lastname; }
27             set { lastname = value; }
28         }
29
30         public float Salary {
31             get { return salary; }
32             set { salary = value; }
33         }
34     }
35 }
```

The mapping file maps the class `Employee` to a table, conveniently named `Employees` could look like Listing 3.3.

Listing 3.3: NHibernate

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <hibernate-mapping xmlns="urn:nhibernate-mapping-2.0" namespace="Firm" assembly="Firm">
3
4      <class name="Employee" table="employees">
5          <id name="EmployeeID" column="employeeid" type="Int32">
6              <generator class="native" />
7          </id>
8
9          <property name="Firstname" column="firstname" type="string"/>
10         <property name="Lastname" column="lastname" type="string"/>
11         <property name="Salary" column"salary" type="float"/>
12     </class>
```

```
13
14 </hibernate-mapping>
```

Several of the elements in the mapping file deal with NHibernate's handling of data, and as we are not using NHibernate itself, we are not interested in these. The focus is on the part of the files that deal with the mapping only.

The following list explains each relevant element in the mapping file and which parts are used in Amigo.

hibernate-mapping: The `hibernate-mapping` element has several, optional, attributes but the ones we use are `assembly` and `namespace` respectively defining where the persistent classes are located and in which namespace they are declared.

class: The `class` element is where the mapping between class and table is defined. The attribute `name` is the name of the class and the attribute `table` is the name of the table. As with the above element the `class` element has additional attributes but they are not useful for Amigo.

id: The `id` element is basically the same as a `property` element, except that i defines the primary key column. The `name` attribute is the name of the property in the class that holds the unique identifier, in the example it is `employeeid`. The `column` attribute is the corresponding column. The `type` indicates the type. Inside the `id` element there is a `generator` element which indicates which algorithm NHibernate uses to generate unique ids.

property: The `property` element is where mapping between properties of a class is mapped to a column in a table. The `name` attribute is the name of the property and the `column` attribute is the name of the column. The `type` attribute indicates the type. In the example it is a `string`.

3.8 Summary

This chapter described the language design of Amigo. First, a number of limitations was established in order for us to keep focus on what is most important at this time; namely designing a safe query language that serves both the object oriented world as well as the relational. The functionality of Amigo was presented in an exemplified manner. The main concept in

Amigo is filters and filter blocks. These make up what corresponds to traditional SQL queries. Other functionality of Amigo include the use of general purpose constructs such as `if-else` and `for` statements.

The full syntax of Amigo was given in EBNF form, and the semantics of each of the language constructs was described in an informal manner. Furthermore, the type system, and especially the type inference mechanisms of Amigo, was presented, along with how the mapping between database and language is handled.

Chapter 4

Implementation

In the previous chapters, we have described a number of inspirational sources for Amigo, and the design for the language. Although we are very focused on creating a strong language design for Amigo in this project, we also feel it is important to implement that design in the form of a working compiler. This enables us to analyse whether the language design is actually a good practical solution. Furthermore, others are able to test and try out the language and its features, to give us an idea of how programmers will take to the language.

This chapter describes the implementation of an Amigo compiler for the Microsoft .NET platform. The software development kit for .NET includes a flexible and easy to use reflection API for generating native .NET intermediate language, which is used in the Amigo code generation. Furthermore, the .NET platform has some advantages when dealing with databases that, for instance, the Java JVM does not. The notion of null-able types, for instance, is a feature that can be very useful in this case, since values in the database are often null no matter what kind of type is in question.

The chapter is divided in two major sections. In the first section we describe the structure of the compiler, and how the syntactical and contextual analysis works. In the other section the code generation from Amigo to SQL and Common Intermediate Language (CIL) for the .NET platform is thoroughly explained.

4.1 Compiler Structure

In order to get an understanding of how the Amigo compiler is structured, this section provides a detailed description of the various aspects of the im-

plementation of the compiler. First, the parser generator used is described followed by an explanation of the concrete syntax tree, that makes use of the visitor pattern [21]. General type checking and how the Hibernate mapping files are used to type check database references is described last.

4.1.1 Coco/R

Implementing a full programming language compiler is normally a large task. Many issues needs consideration and attention. However, some parts of the compiler can be more or less automatically generated by using specialised tools. In the implementation of the Amigo compiler, a tool for generating the scanner and parser parts of the compiler is used. Several such tools exist, but we have chosen to use a parser generator called Coco/R [7] for this particular project.

Coco/R takes as input an attributed grammar on Extended Backus-Naur Form (EBNF) of a source language and generates a scanner and a recursive descent parser. Coco/R supports multi-symbol lookahead, so LL(1) conflicts can be solved by looking k symbols ahead, making Coco/R accept any correctly formed LL(k) grammars.

The choice for using Coco/R as the parser generator for the Amigo compiler, is based on previous experiences with a number of other parser generators, as well as an investigation of Coco/R itself. In the LIP project, we used the parser generators SableCC [12] and Jay [9], neither of which suited our needs very well, and sported various problems during implementation. Coco/R is a lean and simple tool, and supports generating C# based parsers, which is needed in this project.

The language for describing Coco/R grammars is called Cocol/R. This language enables the programmer to specify the grammar for a language in EBNF form, symbol attributes, and semantic actions. Attributes are used to specify the parameters of a production. Semantic actions are inserted directly in the generated parser at the position of the production. To illustrate how language grammars look in Cocol/R we present an example production from the Amigo grammar. This production is used when declaring a filter:

```
Filter<out e.Filter f>          (. string name; string mod;
                                   e.FilterBlock fb;
                                   e.VariableList pl; .)
=
  Modifier<out mod>
```

```
var
Identifier<out name>        (. f = new e.Filter(name, mod); .)
lparen
[ ParamList<out pl>         (. f.Parameters = pl; .)
]
rparen
lcurly
 Block<out fb>             (. f.Blocks.Add(fb); .)

rcurly
.
```

Initially, the example may look a little confusing, but if we look at the EBNF version of the example it quickly becomes apparent what the meaning of it is:

```
Filter = Modifier var Identifier lparen [ ParamList ] rparen
         lcurly  Block  rcurly
```

In the production, symbols that starts with a capital letter refers to other productions, and symbols starting with a lower case letter refers to previously defined tokens. In the Cocol/R syntax for EBNF, a `[ .. ]` means that something is optional. { .. } means zero or more repetitions, and `( .. )` simply groups alternatives whereas `|` separates alternatives.

Attributes are defined inside `< .. >`. They represent the input or output parameters of the symbol for which they are specified. In the above example, only output parameters of other productions are used.

Semantic actions are defined inside `(. .. .)`. Since the semantic actions are directly inserted into the generated parser method for the production, any code that is valid in the host language of Coco/R itself, in our case C#, can be specified here. This means that all code for, for instance, contextual analysis could be inserted in the semantic actions. However, this will almost certainly lead to a huge amount of code within the grammar file. In order to keep the Amigo grammar file as simple as possible, we have decided to create a tree representation of the parsed Amigo source code, the construction of which is what the semantic actions are used for. This means, that the output of the parser generated by Coco/R is a tree representation of the

parsed Amigo source code. We will get more thoroughly into this concrete syntax tree in the following section.

The full Coco/R grammar for the Amigo language can be found in Appendix A.

4.1.2 Concrete Syntax Tree

As mentioned in the previous section, the output of the parser of the Amigo compiler is a tree representation of the parsed source code. This concrete syntax tree is used by the compiler in the contextual analysis of the source code. Using the visitor design pattern, we can easily traverse the tree for performing the contextual analysis and code generation.

Each node in the tree is a representation of the corresponding production in the grammar - at least where such a representation is necessary. We refer to these representations as *elements*. The elements of the Amigo compiler are placed in a library called `Elements` which is referenced from the core compiler. The library consists of several classes that represent each of the elements. An example of an element class can be seen in Listing 4.1.

Listing 4.1: Filter element

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace Amigo.Elements {
6    public class Filter : Element {
7      public override void Accept(Visitor v) {
8        v.VisitIn(this);
9        foreach(FilterBlock fb in Blocks) {
10         fb.Accept(v);
11       }
12       foreach(Variable variable in Parameters) {
13         variable.Accept(v);
14       }
15       v.VisitOut(this);
16     }
17
18     private List<FilterBlock> _Blocks;
19     private string _Name;
20     private string _Modifier;
21     private List<Variable> _Parameters;
22
23     public string Name {
24       get { return _Name; }
25     }
26
27     public string Modifier {
```

```
28          get { return _Modifier; }
29        }
30
31        public List<FilterBlock> Blocks {
32          get { return _Blocks;}
33        }
34
35        public List<Variable> Parameters {
36          get { return _Parameters; }
37          set { _Parameters = value; }
38        }
39
40        public Filter(string _name, string _modifier) {
41          _Blocks = new List<FilterBlock >();
42          _Parameters = new List<Variable >();
43          _Modifier = _modifier;
44          _Name = _name;
45        }
46      }
47    }
```

The element in Listing 4.1 corresponds to the production shown in the example in the previous section. As can be seen, there are a number of properties specific to the element defined in the class. The value of these properties are set during parsing of Amigo source code. If a property is supposed to hold an arbitrary number of other elements, the elements are added to a generic list that has the type of those elements. Figure 4.1 shows a class diagram of the entire `Elements` library.

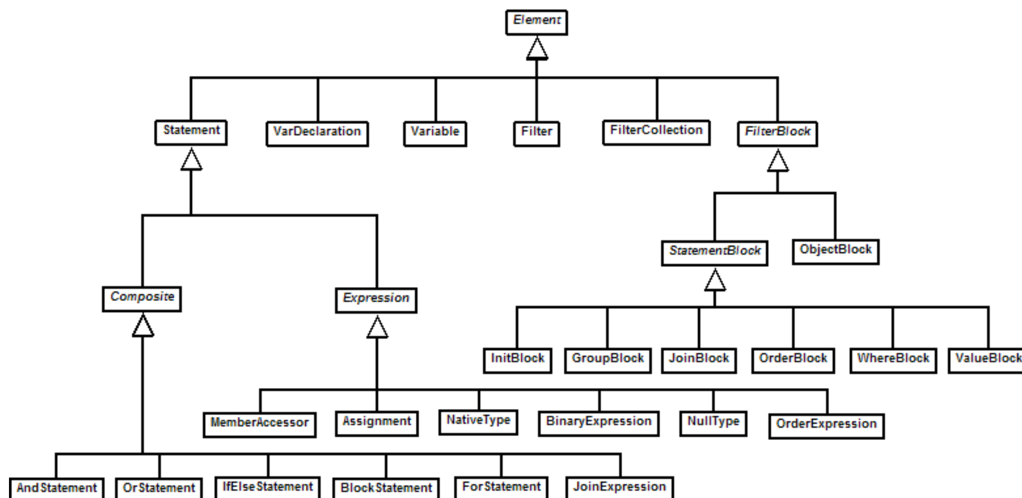


Figure 4.1: Class diagram for the Elements library

All elements inherit directly or indirectly from the abstract `Element` class, that holds an abstract `Accept()` method. This method is overridden in every element in order for the visitor design pattern to be able to visit all the elements of the tree. Each element is visited twice; once before all of its child elements are visited, and once after. This provides us with a little more flexibility when using the visitor. Every time the tree is traversed, once in the contextual analysis, and once in the code generation phase, a visitor is used. The class `ContextAnalyzer` in the *ContextualAnalyzer* library implements the `IVisitor` interface, meaning that for each call to an element's `Accept()` method, the corresponding visit methods of the element are called in the `ContextAnalyzer` class. The same applies for the `CodeGenerator` class.

The use of the visitor pattern, and several classes to create the concrete syntax tree may not be the optimal solution - performance wise. We could possibly save a pass over the tree by leaving more code in the semantic actions of the Coco/R grammar, for instance for contextual analysis. However, we have decided on this solution in order to keep the compiler as easy to understand and explain as possible, and since we are not concerned with performance in this particular project, the solution does not in any way conflict with the initial goals for this project.

4.1.3 Type Checking

In Section 3.6 we described the reasoning behind the type system of Amigo. Now we discuss how it is implemented in the actual Amigo compiler.

Since Amigo source code is parsed into a concrete syntax tree, as explained earlier, all necessary parts of the source code is represented as elements in the tree. The basic types of Amigo are represented by the element class `NativeType`. This class has properties that are set according to which basic type is currently in question. Variables are represented by the element class `Variable`. The type and value of the variable is specified within the class. All variables are declared in some scope. If a variable is declared within the `init`-block, it is available in the entire filter, whereas a variable declared anywhere else is only available from within its own scope. In order to keep track of variables and their scope levels, each variable, upon its declaration, is entered into the `SymbolTable` class. This class is a part of the *ContextualAnalyzer* library. When the contextual analyser, while visiting the elements of the concrete syntax tree, leaves a scope, the variables in that scope are removed from the `SymbolTable` so that they are unavailable

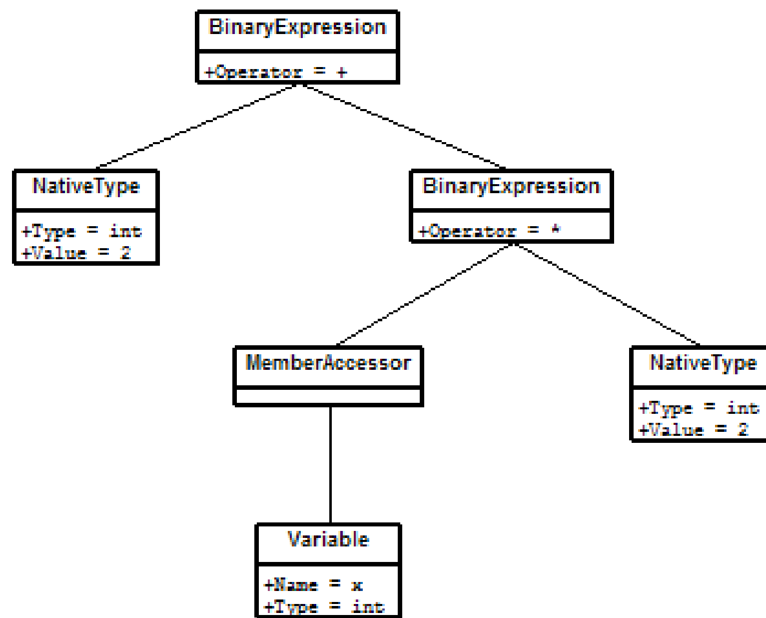


Figure 4.2: A subtree in the concrete syntax tree

outside the scope in which they have been declared.

As can be seen in Figure 4.1, Amigo expressions are represented in the syntax tree by a group of elements that are all specialisations of the abstract `Expression` class. This class has a non-abstract property that holds the type of the expression. Imagine, that the following variable declaration is found inside an `init`-block in an Amigo source file:

```
int i = 2 + x * 10;
```

This simple example corresponds to the subtree seen in Figure 4.2 of the concrete syntax tree.

When the contextual analyser checks this variable declaration, it will first check whether all elements of the expression (the part to the right of the assignment operator `=`) are of the same type. For each binary operator in the expression, an instance of the `BinaryExpression` is present in the tree, with a left and right hand side, and a property that holds the operator itself. Because of the Amigo precedence rules, the subexpression `x * 10` is checked first. Native types are easily checked, since their type is already known. Variables, or member accessors, are looked up in the symbol table

to make sure that they have been declared, and if that is the case, the type of the variable is thus known from its declaration. It is important to note, that only member accessors that are actually variables are looked up in the symbol table. If the member accessor is a call to another filter, or a reference to a field in the database schema, appropriate action is taken to make sure that the member accessor is valid. Filter calls are looked up in a table that holds information on all declared filters, and schema field references are looked up in the mapping file. The latter is further explained in the next section.

Once the contextual analyser has determined the type of the elements of the binary expression, the two types are compared, and the type of the entire binary expression is set to the type of its elements if they are in fact the same. Next, the same apply for the binary expression `2 + (x * 10)`. Since the contextual analyser already know the type of the right hand side of the expression, it merely has to compare that to the type of the left hand side, which is easily determined since it is a native type. Now, the type of the full expression has been determined and it can be compared to the type specified in the variable declaration. If they are not the same, a type error occurs.

4.1.4 Mapping

As was mentioned in Section 3.7, we use NHibernate mapping files to represent the database schema. In order to avoid type errors at runtime, when referencing fields in the database schema, all of those references are type checked at compile time. Recall, that the element class `MemberAccessor` could represent both internally declared variables as well as external schema references. The contextual analyser will determine which of the two are applicable in the given case by first looking up whether the member accessor is a variable present in the symbol table. If that is not the case, it is looked up in the schema mapping.

In order to ease looking up the member accessor in the schema, a library for representing the schema in a simplified tree version of the NHibernate mapping file has been implemented. This library is called `MappingTree` and is referenced from the `ContextualAnalyzer` library. In Section 3.7 we argued that not all parts of a Hibernate mapping file is useful to us. Therefore, the elements that make out the mapping tree are simply `MappingClass`, `MappingProperty`, and `MappingRelation`. We refer to these classes as *mapping classes*. Furthermore, a `Root` class is present. The main responsibility

of this class is to actually construct the tree by reading and parsing the XML that make up the mapping file.

Each of the mapping classes has properties that hold the appropriate information for the particular class. For instance, `MappingClass` represents the mapping between a class in the application and its corresponding table in the database. This means that information about the class name, table name, and properties of that class is held by this mapping class.

4.2 Code Generation

The main task of generating the code from the syntax tree, which is built using the semantic actions in the Coco/R grammar and validated using the contextual analyser, is to translate the query expressed in Amigo to SQL commands that can be nested in Common Language Runtime objects. These objects can then be invoked from any language running on the Common Language Runtime.

4.2.1 SQL Generation

As covered in the previous section we use a concrete syntax tree, which is traversed using the visitor pattern, to perform contextual analysis. This approach is also used to generate both the SQL code and to produce an assembly, containing the filters in the form of instance methods that can be invoked from .NET.

The information present in the syntax tree is very detailed and only few decorations of the tree is needed during contextual analysis. However, a few issues regarding types is checked and arithmetic expressions are decorated with their overall result type. This information is very valuable as it guarantees that the SQL code generated will have the correct type information when the SQL statements are constructed.

Basically, a `SELECT` statement in SQL is built of six parts: `SELECT`, `FROM`, `WHERE`, `JOIN`, `ORDER`, `GROUP` and `LIMIT`. Because we mostly build plain SQL statements, a data structure to store these elements during the visitors visit of each filter in a source file is used. When the code-generator visitor leaves the filter element in the tree, the SQL is flattened to an SQL string which is later wrapped in calls to database functionality during the Intermediate Language code generation.

In Amigo, some expressions have a slightly different semantics than would be expected in comparison to most general purpose programming

language. For instance, a list (or array) of values can be compared with a single value in the `where`-block of Amigo. This requires special attention in the SQL generation as this has to be translated to a clause that makes use of the range feature of SQL. So the list is flattened to a comma separated list of its values and is injected into the SQL statements `WHERE` part as an expression on the form:

```
... WHERE colX IN (1,2,3,4) ...
```

Another problem that has to be resolved when generating the SQL code is to prefix all columns with their table names, so that column names that are used in several tables, which are part of a query, can be distinguished. This is not a straight forward task as we, in case of many-to-one relationships, only have this data available on one end of the relationship in the mapping files. And as the complete mapping tree needs to be loaded to make sure that the other end of a relationship is available, the task of assigning table names to all fields is done by looking up the table names in the mapping tree in the situation that it is not available in the mapping for a relation. A fragment of two class mappings is shown in Listing 4.2, which illustrates the problem with the missing information in the mapping for the `Orders` class.

Listing 4.2: Relationship mapping

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <hibernate-mapping xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4                     xmlns="urn:nhibernate-mapping-2.0">
5    <class name="Firm.Customers, Firm" table="Customers" lazy="false">
6      ....
7      <bag name="Orders" access="property" table="Orders" lazy="false">
8        <key column="CustomerId" />
9        <one-to-many class="Firm.Orders, Firm" />
10     </bag>
11   </class>
12
13   <class name="Firm.Order, Firm" table="Orders" lazy="false">
14     .....
15     <many-to-one name="Customer" access="property"
16         class="Firm.Customers, Firm" column="CustomerId" />
17   </class>
```

The generation of the SQL statement itself is straight forward, provided that the type system and syntax tree is correct. `SELECT` statements have a relatively fixed structure and are, as mentioned, composed of keyword-groups that describe the separate functionalities of the statement. This is the case in almost any scenario.

4.2.2 Intermediate Language

The generation of Intermediate Language (IL) [2] code for the .NET platform is not as straight forward as the generation of the SQL statement. The Common Language Runtime is a stack based virtual machine, and even though the intermediate language is powerful, it is still often more like programming low level assembly code than a general purpose programming language. The `System.Reflection` API provided by the .NET framework facilitates the generation of IL code, without having to concentrate on low level aspects of the code generation. This API includes builder classes for building specific parts of IL code, and provides easy generation of common structures like classes, constructors, methods etc. A diagram showing the builder classes can be seen in Figure 4.3.

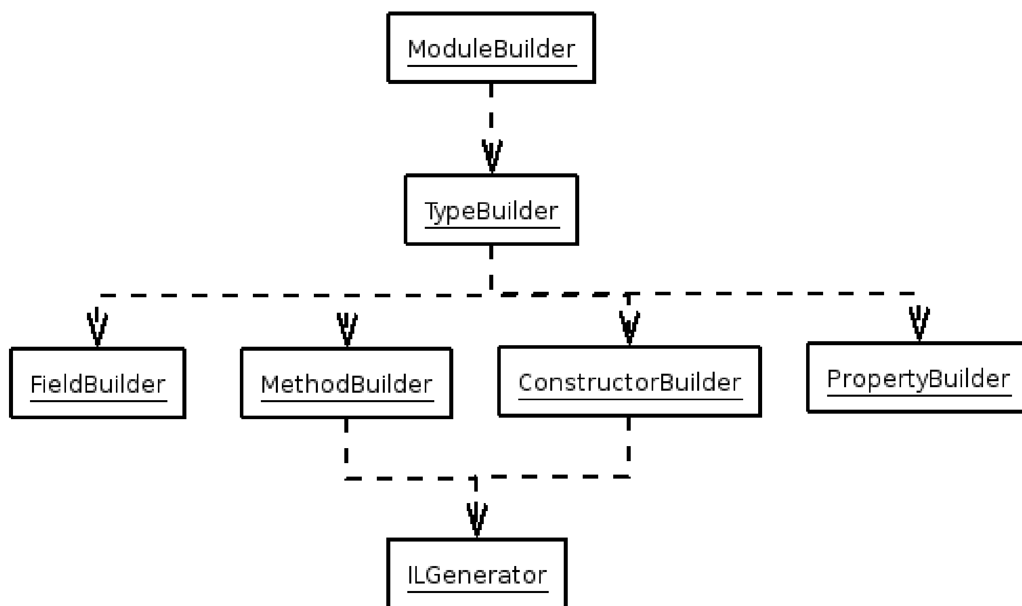


Figure 4.3: Structure of builders

The builder classes are structured in a Factory pattern. As Figure 4.3 shows, one builder class is used to create one or more builders for subtypes

and it is possible to gain access to a `ILGenerator` that generates the body of either a method or a constructor. The `ILGenerator` has methods for emitting code to a stream of IL operations. This is done either by calling the `Emit` method with an operand code found in the `OpCodes` enumeration. The `Emit` method is available in several overloads. Depending on the demands for the operand code, multiple arguments can be sent to, for instance, describe what input is present on the stack.

Additionally, the `ILGenerator` have a collection of methods for more advanced tasks. Some of these are listed below:

- BeginScope: opens a new scope level for variables
- EndScope: ends the current scope
- DeclareLocal: declares a local variable in the current scope

In the Amigo compiler, the `ModuleBuilder` and `TypeBuilder` instances are used in the constructor on the `CodeGenerator` visitor. When the visitor enters a `Filter` element of the syntax tree, a new `MethodBuilder` is created and its `ILGenerator` is stored as a variable on the `CodeGenerator` visitor, so that it is accessible from all visit methods that might need to generate IL code. When the visitor leaves the `Filter` element, the last IL code for executing the SQL statement is generated using the `ILGenerator`. The generated code is simply a `Connection` object and a `Command` object, that makes it possible to connect to the back-end database. Depending on the contents of the filter, the method that is built returns either a `DataTable` or an `object`.

To illustrate the use of the above builder classes and the `ILGenerators` received from them, a small example that generates a class with a single method is shown in Listing 4.3

Listing 4.3: Buidling an assembly using TypeBuilder, ModuleBuilder etc.

```
1  //Define class for holding name of the assembly
2  //and assign a name to its Name property
3  AssemblyName an = AssemblyName();
4  an.Name = "myAssemblyName";
5
6  //Retrieve the AppDomain for the current Thread
7  AppDomain appdom = AppDomain.CurrentDomain;
8
9  //Use the AppDomain to create a dynamic assembly and retrieve its
10 //AssemblyBuilder, as an argument it takes an AssemblyName and an
11 //indication if the assembly should be saved in memory or on disk
12 AssemblyBuilder ab = appdom.DefineDynamicAssembly(an, AssemblyBuilderAccess.Save);
13
```

```
14 //We use the AssemblyBuilder to retrieve a ModelBuilder for the
15 //Assembly and specify the name of the module, the name of the file it
16 //should be saved in and whether it shall automatically emit symbolinfo(names)
17 //etc.
18 ModuleBuilder mb = ab.DefineDynamicModule(an.Name, an.Name + ".dll", true);
19
20 //We use the ModuleBuilder to define a type in the Module and retrieve
21 //a TypeBuilder, the arguments specify the name and TypeAttributes in our
22 //case we define a public class
23 TypeBuilder tb =
24     mb.DefineType("MyNamespace.MyClass",
25       TypeAttributes.Public|TypeAttributes.Class);
26
27 //We use the TypeBuilder to define a method in the Type and retrieve a
28 //MethodBuilder, thte arguments specify the name of the method, and
29 //methodattributes which could be if its static etc. third we define the
30 // return type of the method and at last a list with types of the parameters
31 MethodBuilder meb =
32     tb.DefineMethod("MethodName", MethodAttributes.Public,
33           typeof(string), new Type[]{typeof(int)});
34
35 //We retrieve an ILGenerator from the MethodBuilder which we can use to
36 //emit IL code to the body of the method
37 ILGenerator ilgen = meb.GetILGenerator();
38
39 //We declare a local variable in the method body
40 ilgen.DeclareLocal(typeof(string));
41
42 //A string "Hello world" is loaded on top of the stack
43 ilgen.Emit(OpCodes.Ldstr, "Hello world");
44
45 //And the top element of the stack(our string) is stored to
46 //the variable with index 0 which we just decared above
47 ilgen.Emit(OpCodes.Stloc\_0);
48
49 //The Type is created using the typebuilder
50 tb.CreateType();
51
52 //The asssembly is saved to disk
53 ab.Save(an.Name + ".dll");
```

4.3 Summary

This chapter described the implementation of Amigo. First, Coco/R was introduced. Coco/R takes a grammar on Extended Backus Naur Form (EBNF) and produces a scanner and recursive descent parser. The choice of Coco/R was based on bad experience with other parser generators, mainly from the LIP project. The language for describing grammars in Coco/R, Cocol/R was described. The output from the parser in the Amigo compiler is a tree representation of the parsed source code. The visitor pattern allows

us to easily traverse the tree to perform the contextual analysis and code generation. Then the type checking was described. Following this was a description of the mapping which is done using Hibernate mapping files representing the database schema. Finally code generation was described. The description explained how Amigo code is translated into SQL queries, as well as how these queries are inserted in an assembly of methods, that each corresponds to a filter. These methods can be called from any .NET based language that can reference the assembly.

Chapter 5

Test

When working with programming languages and compilers, creating a good syntax or a well performing compiler are not the only things to worry about. There are several aspects that have to be dealt with, one of which is testing. It is very important to make sure that the language and compiler is working in a correct manner. Since a programming language is used to write an application, and a compiler is used to translate that application into running code, there is little room for errors in the compilation process. If the generated code is erroneous, by fault of the compiler, it will cause confusion and irritation for the programmer using the language and compiler to create his application. It will most likely not be clear that an error is an error in the compiler and not in the source code of the application.

When we talk about software testing we talk about testing strategies and testing tactics. Strategies describe how testing is conducted; if the entire program is tested or only parts of it; if the tests should be run every day or only when the entire program is finished. Tactics describe the specific tests on a technical level.

There are a plethora of testing strategies and many theories on how to vary or combine them. Most involve testing smaller parts on their own, testing the program as a whole which basically means testing if the smaller parts work together, testing the program on the system on which it will be used, and validating the program. Some strategies do these sequentially but most use some sort of incremental test strategy. A software development process usually starts with a customer expressing requirements that the final program should meet. Then a rough model of the entire program is designed. This is refined and interfaces are generated. Then classes are designed and implemented one by one. Depending on time and money each

class or unit is tested when it is done. In some cases the entire system is regularly tested using dummy classes as substitutions for those that have not been implemented yet. As soon as parts of the program works, validation begins. This is the part where the customer evaluates the program against the requirements, which are not specific enough to account for every detail. At some point, depending on the specific task, system testing begins. Some developers choose to perform these tests only once, or as few times as possible. They will most likely only test when the entire program is done, correct the errors and test again. This is not advisable, especially for larger projects, as debugging errors are much harder at this point. A more advisable approach is one where all tests are performed several times throughout the process. Again there are variations of this approach, and most development teams will customise a strategy to suit their particular situation.

Testing Amigo entails testing whether the compiler works as it should, and testing whether the Amigo language is useful. To test whether the compiler works as intended, we use unit testing and test it with a number of example filters against the scenario specified in Section 3.2. To test how programmers take to the language, a validation test is performed and described. The test subjects for this test have not in any way been involved in the development of Amigo.

5.1 Code Testing

We have chosen to perform unit tests based on the expected output from the database. That is, the data output of an SQL query is compared to the data output of the corresponding Amigo code. To perform this operation we have used the NUnit test framework [11], along with a few helper classes to execute SQL on the DBMS. The DBMS we have chosen to use for the test is PostgreSQL [13]. This particular DBMS is very feature rich, and we are familiar with it from previous projects.

5.1.1 Unit Tests

Unit tests are performed by testing the result of a filter against a sample database to verify that the filter returns the rows/values from the database that are expected. As these tests work by comparing two data tables, one generated by the Amigo query and one by the SQL statement, we have written some custom code to perform a comparison between two data tables,

disregarding issues like column aliases, as these are generated by the Amigo compiler. The comparison is done solely on the data and not on meta-data such as headers and so forth.

To integrate the unit test with an existing test tool, we have decided to execute them using the NUnit[11] testing framework. We have implemented a method called `DataTableComparer` that compares two data tables by taking a `DataTable` and an SQL query string as input arguments. The `DataTable` is the result of an Amigo query. The SQL query is executed on the DBMS and the resulting `DataTable` is compared to that of the Amigo query. If these contain the same values the method returns true, otherwise false. This method is used in ordinary unit test setups.

```
[Test]
public void TestSimpleEmployeeQuery() {
    Compiler c = new Compiler("inputfile.amigo");
    Assert.AreEqual(true, DataTableComparer.AreEqual(
        c.Queries[0].DataTable,
        "SELECT EmployeeID, Name, ... FROM Employees"
    );
}
```

In the example, all mapped columns in the database must be specified in the `SELECT` statement, so that the resulting data is consistent with what Amigo generates. An error in the output test will be presented in the NUnit GUI so that data that is not returned in the expected format is found and can be debugged.

The approach outlined above ensures that all compiler tests that fail are present in a single tool, integrated in modern development tools and hence is visible for us during development of the Amigo compiler.

5.2 Validation

Normally, validation testing is performed against a set of requirements from the customer. But in this case, there are only potential customers and thus no explicit requirements. The requirements are defined by the developers and validation testing of Amigo is therefore also a test of whether the requirements defined by the developers match those of the potential customers. One of the requirements was that Amigo should be easy to under-

stand for any programmer with experience in object oriented programming and relational databases. The test subject is given a short Amigo manual; see Appendix B. He is then presented with nine exercises. The exercises are based on the scenario from Section 3.2. Each exercise is formulated in plain English and the test subject is asked to write Amigo code which will solve each exercise. An example of such an exercise could be:

"Write one or more filters which will retrieve all employees with a salary of more than 20.000."

The Amigo code which we expect the test subject to write would look like this:

```
public var HighPaidEmployees() {
    object:
        Employee;
    where:
        Employee.Salary > 20000;
}
```

The exercises are all presented in Appendix C in the same form as the test subject is presented with. The queries are fairly simple, meaning that they do not involve more than two tables. We expect to learn if the block structures of the Amigo filters make sense to users. In other words, we are interested in testing the basic concepts of Amigo. The Amigo code, that the test subject writes, is evaluated in two ways. First, did he manage to write code that has the result which the exercise stated. Second, is the solution the same as we expected. The first tells us whether some data retrieval problems are more difficult to translate into Amigo code than others. The second tells us if there are more than one way of doing the same thing, and which way is the better or more intuitive choice. The test subjects are given 45 minutes to read the manual and complete as many exercises, out of a total of nine, as possible.

5.2.1 Results

We present the results for each test subject and after wards a conclusion.

Test subject 1

Subject 1 managed to complete the first six exercises. He rated the completed exercises as follows:

Exercise	Difficulty Rating
1	1
2	2
3	3
4	2
5	5
6	1

As can be seen, exercise 5 was rated more difficult than the others. In this exercise, the solution calls for the use of joins. The test subject commented on this in the overall evaluation of Amigo. He wrote that he likes the block structure but that he did not like the join construct. He felt it was conceptually different than the other blocks and that it would take longer to get used to. Overall, he felt that Amigo was well structured and fairly easy to use.

Test subject 2

Subject 2 also managed to complete the first six exercises. He rated the completed exercises as follows:

Exercise	Difficulty Rating
1	1
2	1
3	1
4	3
5	3
6	1

Once again the same pattern is seen, although with lower numbers. This test subject did not find joins as problematic as test subject 1, which is also reflected in his evaluation. He commented that Amigo seems very intuitive in the way queries are formed. He pointed out, however, that it requires a different approach than SQL. He especially liked the way Amigo handles joins which is better than having to explicitly specify primary keys

and foreign keys. This is in contrast to the opinion of test subject 1. Test subject 2 found that the exercises very relatively easy and simple. He would have liked to try Amigo in more complex scenarios. He finally states that Amigo certainly bridges the gap between the object oriented and relational worlds, and that further development could prove very interesting.

Conclusion

During the tests, both of the test subjects seemed to think in terms of tables and SQL and not in terms objects. They had trouble adjusting to the idea of not specifying primary and foreign keys relationships when joining tables. The fact that both subjects read the manual and completed six out nine exercises in 45 minutes, we believe reaffirms that Amigo is easy to learn. As test subject 2 pointed out, the exercises and scenario were simple. The test then confirms only that Amigo is easy to learn for simple tasks. It is yet to be proved that it is also easy for complex tasks and scenarios. A longer test, maybe several days, would perhaps make them think in terms of objects more than in terms of tables. All in all, the outcome was as we had hoped. Both test subjects thought that Amigo was interesting and worth developing further. They wrote solutions that were very close to the ones we thought they would write. This indicates that it is obvious how to use Amigo to solve different types of tasks. The join construct could be worth re-investigating and possibly redesign its syntax. But the fact that the two subjects did not agree on this means that further testing has to be done.

5.3 Summary

This chapter described the testing of Amigo. First, different approaches to testing was discussed. Then a description of which methods to test Amigo was presented. Unit tests were chosen to test code correctness, and validation test was chosen to test what programmers thought of Amigo. Both tests used the example scenario defined in Section 3.2. The unit tests were performed on database output. That is, compare the output of an Amigo query and the output of a corresponding SQL query.

In the validation test, two test subject were presented with an Amigo language manual and an exercise set with nine exercises. They got 45 minutes to read the manual and complete as many exercises as possible. The result was, that both subjects found that Amigo was intuitive and easy to use. One subject however, thought that the join construct was difficult

to understand and use. Finally, it was concluded that the simple validation test was a success, but also that even more testing is necessary.

CHAPTER 6

Related Work

In this chapter we briefly discuss some of the major contributers to work related to the area of which Amigo is part. More specifically, Language Integrated Query, a technology being developed by Microsoft Corporation, and Hibernate are discussed. We describe how they relate to the Amigo project as well as compare strong and weak points of the technologies in relation to Amigo.

6.1 Language Integrated Query

Language Integrated Query (LINQ) [10] is a set of features for the C# and VB.NET languages for the Microsoft .NET platform. The features are being developed by Microsoft itself, and are planned to be a part of C# 3.0 [15]. LINQ provides the programmer with the ability to query any collection of data in much the same manner as using SQL queries to query a database. A number of extensions are added to the language and compiler in order to make LINQ work. These extensions include: extension methods, anonymous types, type inference, and lambda expressions. The extensions are, among other things, used to implement a set of standard query operators, that are added to the generic type `IEnumerable` (by using extension methods) which all generic collections inherit from. The standard query operators makes it possible to query the collections. The C# 3.0 compiler adds syntactic sugar to the language in order to make the use of the standard query operators look more declarative - or more SQL-like.

The LINQ extensions are used to implement DLINQ, which in turn is used to query relational data. This means, that a database query can be expressed directly in the language, by using the standard query operators.

6.1.1 LINQ vs. Amigo

The philosophy behind the LINQ project - and especially DLINQ - is quite similar to that of Amigo. Furthermore, both projects operate on the same software platform. This could possibly mean direct competition between the two. However, Amigo does not require any extensions to neither the C# language nor the C# compiler - let alone the .NET runtime platform itself. To that end, the Amigo compiler could relatively easily be reimplemented to use LINQ, and thus benefit from the same features as DLINQ, while still using the Amigo language syntax and features.

The look-and-feel of LINQ is very similar to SQL - at least while using the syntactic sugar versions of the standard query operators. This kind of syntax will most likely fall natural to most programmers, who are already familiar with SQL. Amigo, on the other hand does not use a SQL-like syntax, which possibly could scare some programmers away from the language. We do believe however, that the Amigo syntax provides a simple way of expressing queries, that would be very complex to express in a SQL-like language. The general expressiveness of LINQ and Amigo, however, is very similar - although LINQ provides means to query all kinds of collections, while Amigo can only query relational databases.

6.2 NHibernate

NHibernate is an object/relational mapper for the .NET platform. As such,it maps data from the object model to the relational model. NHibernate also has a query language, HQL as described in section 2.4, making data retrieval easier. The idea behind NHibernate is that the programmer for the most part can ignore the database a concentrate on NHibernate and the application. There are several tools to help in generating the XML mapping files from the object model and the database schema from the mapping files. The people behind NHibernate does recommend that the files should be examined by a person to ensure the most effective and logical mapping. There are tools which can do the reverse which is helpful the database already exists.

NHibernate acts as a layer between .NET applications and the underlying database. Any object which is persisted go through the layer and every query and also pass through the layer. If the programmer follows coding recommendations, NHibernate handles the generation of unique id's for all

persisted classes and subsequently lets the programmer retrieve a specific class if the id is known.

NHibernate uses several methods to enhance performance. The first is cache which holds retrieved classes. The second is the option to lazy-load classes or collections of classes. An example where both methods come into play could be a situation where the programmer has to perform something on a number of objects which cannot be expressed in a query. NHibernate allows the programmer to return a query as a `System.Collections.IEnumerable`. Doing this only returns the ids of the objects and the `IEnumerable` lets the programmer iterate through them. The objects are loaded when the enumerator gets to them. If only half the objects are used, then only half is loaded. If any of the object are in cache they are not retrieved from the database.

6.2.1 NHibernate vs. Amigo

In the analysis we have already discussed the pros and cons of HQL but we deliberately did not discuss the rest of NHibernate. It works very well and judging by the amount of downloads from *sourceforge.com* it is very popular. We think Amigo is easier to use than HQL and that it supports the programmer more than HQL does. HQL does add many new functions to SQL but it still suffers from the very problems that Amigo tries to solve.

The interesting question is whether Amigo could be implemented as a part of NHibernate. As we argued above in the LINQ section on unifying Amigo and LINQ, Amigo could also be made to use NHibernate. Initially, Amigo could be reimplemented to simply return HQL instead of SQL, but it is also a possibility to bypass the HQL and let Amigo use NHibernate's O/R-mapping features directly.

Chapter 7

Evaluation

In this chapter we evaluate the Amigo language, the report, the process, and the tools used to develop Amigo. The chapter starts with an evaluation of the two types of tests we performed, beginning with the validation test and then the technical test. Then we evaluate the functionality of Amigo, whether we met the requirements we stated in the problem statement in Section 1.1. Following this, we evaluate the requirements themselves, determining whether or not they were the right ones for this project. Next, an evaluation of Amigo itself is presented. In Section 3.1 we stated some limitations and it is important to evaluate them to determine if they were the right ones and what impact they have had on Amigo. Finally we evaluate the tools we have used to develop Amigo.

Several times in this section we use the term useful. We define useful as the ability for programmers to incorporate Amigo in their software development environment, or chose Amigo over an existing solution that accomplishes the same task.

7.1 Validation Test

In Section 5.2 we described how we had devised a set of exercises which was used in a test with two test subjects. The test had three aspects. The first was the actual exercises where we tested if the subjects were able to write Amigo code. Second we asked them to rate the difficulty of each exercise. And third, we asked them to evaluate Amigo.

The exercises increased in difficulty and we expected the Amigo code they wrote to give us an indication of what types of tasks where particularly difficult. When we devised the exercises we also discussed what solutions

we expected the subjects write. It would help to have a preconceived idea of how we believed certain tasks should be accomplished in Amigo. If the subjects chose to create different solutions then perhaps the Amigo syntax had to be revised, or maybe the manual needed improvement.

We selected two persons as test subjects. This is too few to create any type of meaningful statistics. This, however, was not the goal of the test. The reason why we asked them to rate the difficulty of each exercise was to add another way of determining whether certain task were harder to accomplish than others. The exercises were formulated in a way that resembles everyday programming tasks. If they both rated three out of five on each exercise except one were they both rate five out of five it would mean that that particular type of task is difficult to carry out in Amigo and something has to be changed to remedy this fact.

In the final question of the test, the test subjects were asked to evaluate Amigo. We expected to get information about how intuitive it was to use, and aspects of that sort. And of course the verdict whether they thought it was a something they would use or not. In our view, this was the most beneficial part of the test, as we got an idea of what the subjects thought about the language as a whole.

We gave each test subject 45 minutes to read the manual and complete the exercises. In retrospect this was perhaps not enough time, as they both only managed to complete six of the nine exercises. An hour or more would have let them finish all the exercises, or at least try to finish them. In Section 5.2.1 we concluded two things from the test. One, Amigo was intuitive and easy to use, although it takes a little time to fully understand the Amigo train of thought. Second, the join construct may need to be changed, but the subjects differed on this, so additional testing of this is required. We did not get as much information from the exercise ratings as we had hoped. We should have imposed a few criteria for each rating, so that if the subject had no problems solving the exercise and write the correct code in one go, it should be rated at 1. If the subject solved the exercise but had to try several times it would be rate at 2 etc. Overall the test was a success but could have been even more successful.

7.2 Technical Test

The unit tests worked by comparing the results from Amigo code and corresponding SQL statements. This makes finding errors more difficult as

backtracking from the database result to the error in the code generation is difficult. It would be better to test even smaller fragments of Amigo one at a time. The problem is that we generate the code by traversing the syntax tree, and each element in the tree calls methods on its children. We cannot jump into middle of the tree and generate code from that point. Based on this technical limitation, testing based on database results is a tolerable solution.

7.3 Analysed Technologies

In Chapter 2 we investigated a number of technologies that we found interesting in relation to the development of Amigo. We analysed the querying features of HQL, T-SQL, PL/SQL, and HaskellDB. The primary goal of this analysis was to gain inspiration and further knowledge of already existing technologies. We feel, that this investigation was successful in the way that we did learn a number of things in the process. These things in some cases did indeed serve as inspiration for the further development of Amigo.

The one technology that stood out in the investigation was HaskellDB. The reason for investigating HaskellDB was that the reasoning and arguments behind this technology was very similar to those of Amigo. However, the actual implementation of HaskellDB and the practical side of this solution did not have a very big impact on the design and implementation of Amigo. We still think, that the investigation of HaskellDB served a purpose, as it proved to us that it is possible to create a safe way of interfacing between the relational paradigm and a completely different paradigm - in this case the functional.

7.4 Functionality

In the problem statement in Section 1.1 we stated several goals. In this section we evaluate these goals. One goal was to extend the filter concept, which we developed in the LIP project, to use more RDBMS features. Amigo supports the same types of queries achievable by standard SQL. We even made sure that it will be possible to use RDBMS functions unknown to us. Every call is mapped to a database function — if it exists. Amigo first test if the call is to another filter and the tries to map to a database function.

We also stated that filters should fit in the object oriented paradigm and comply with its model. Our notion of this is that the syntax and semantics of the basic structures from the object oriented paradigm is preserved within the filters. Examples of this are loops, conditional statements and arithmetic expressions which all share the semantics of the corresponding C# constructs. When we introduced features we strove to use known concepts from the object oriented world with a common C# syntax. For instance, a join is performed by applying a binary operator on two types. The block that a filter consists of shares syntax with the goto and case constructs known from C#.

Even though the new constructs introduced use common syntax with existing features in C#, the filter has a mostly declarative body whereas the existing method construct in C# has an imperative body. However the declarative syntax is well suited for the query languages as SQL has shown. This declarative form of filters complies with the goal stated in the problem statement that the relational model is not meant to be abstracted away in Amigo.

The final thing we stated, was that the programmer should be aware of when he is using the database. This goal is trivially solved after having solved the above. All database interaction occurs inside filters and the programmer is fully aware of that.

7.4.1 Project Goals

We have evaluated the project goals in regards to whether we met them or not. Now we have to ask the question, if these were the right goals. The primary goal was to extend the filter concept from LIP to use more RDBMS features.

In our analysis of other technologies, both in this report and in the LIP report, it is apparent that programmers who use relational databases do so because they use the built-in functionality.

Using filters ensures that database related code is not spread out through the other code. This also allows combination of filters - with the use of filter calls - improving the modularity of filters. Furthermore, this helps to reduce the size of filters and should also reduce the overall amount of code, while increasing readability and writeability.

We have been pleased with working with this concept, and we still are confident that filters are a clean and intuitive means of expressing database queries - although there are room for improvements, as our tests have shown.

The second goal was to implement the filter concept as a new query language. This is an obvious goal, as it is the only way to really test the concept in practice. Programmers would have had a difficult time testing the concepts of Amigo if they did have the possibility to get a hands-on feel of it.

The next goal was that Amigo had to support most of the functionality provided by SQL. This goal has been met to some degree. Amigo supports the basic SQL functionality as well as some more advanced features. Furthermore, Amigo includes a few features that are not directly available in SQL. However, there are features in SQL that are not present in Amigo. These features have been left out in this project, as their functionality would not have a significant effect on the Amigo syntax and semantics. The basic idea of a query language is to be able to retrieve data from a relational database, and SQL by itself accomplishes this very well - and is an extremely popular approach among programmers. Therefore, this goal was one that could not be left out, and we are generally pleased with the results of it.

The next goal was that the filters should comply with the object oriented model but the relational database should be visible. The details of this goal was not stated clearly enough when the goal was initially presented. Furthermore, it is arguable whether it is relevant to this particular project. Obviously some notion of object orientation is required in Amigo, since it is meant as a bridging between the object oriented and the relational world. However, there is no need for a complete object model in Amigo until it is integrated in an object oriented host language.

The next goal was again a direct consequence of LIP. The programmer should be aware of when he is using the database and when he is not. LIP handled insertion and updates, in addition to queries. We established that the programmer would feel more in control if insertions, updates, and deletions were explicit commands. Amigo does not perform insertions, updates, and deletes and the goal thus becomes meaningless. This is at least true as long as all query code is inside filters, where it is obvious to the programmer that he is using the database.

We also stated as goals that we needed to analyse existing technologies to establish which features Amigo should support. We also stated that Amigo should be tested both from in a technical test and in a validation test. Analysing existing technologies is almost an unwritten rule when developing new software or concepts. Others may have developed software that has been tested by end-users over a period of time, which is reflected by the software. In other words, analysing other technologies will help to

establish user requirements. Testing is also something that simply must be done, at least the technical test. There is no point in releasing buggy code. Performing a validation test is not a must, but it is very useful and based on the test we performed, a great deal of information is acquired. When you spend a lot of time on a project it is difficult to evaluate the product objectively. An independent tester will spot things you did not think of. We are very pleased with the results of the tests, even if some did not comply with our own thoughts about Amigo. To that end this goal was a good one, but could certainly be expanded at a later time.

7.4.2 Usefulness of Amigo

Most of the goals that were established for Amigo have been met, some to a higher degree than others. Now, the question is if Amigo is good enough to be used by programmers in projects today. A decision has to made about how the language is to be used, and Amigo must be implemented accordingly. We are very confident that Amigo would be practically usable when this is done. The decision is not an obvious one and it would be valuable to do further testing of whether Amigo should be used as a preprocessor or natively integrated into a host language to determine in what form Amigo would be of most value. This means that Amigo in its present form is not fit for use in a professional software development environment. However, the concepts and fundamentals are sound.

7.5 Limitations

In Section 3.1 we stated some limitations and it is important to examine what impact this has had on the outcome of Amigo. First we introduced each limitation along with a few comments and then we discussed the general impact they have had — if any.

We stated focus of the Amigo project was on querying and not on other parts of mapping and persistence. Although we did not focus on these there is nothing in Amigo which prevents them from being implemented. As we pointed out in Chapter 6, Amigo could be used as part of LINQ or Hibernate. We attribute this, in part, to the filter construction which encapsulates every query.

Amigo checks type mapping via the Hibernate mapping files. If Amigo is to be used as part of Hibernate there would be no reason to change this as Hibernate relies on these as well, and in this case the limitation is not a

limitation at all. If Amigo is to be used as part of C# or any other language it is better to let Amigo communicate directly with the database, as making mapping files only impose extra work for the programmer.

We stated that we would not focus on integrating Amigo in a general purpose language. This is true in the sense that we have not focused on how to most effectively integrate Amigo in for instance C#. We have argued in e.g. Chapter 6 that it would be possible to do so. This issue is discussed further in Chapter 8.

Based on the experience with LIP, we expected inheritance and polymorphism to be an area requiring a great amount of work and thus best not have that as primary focus. It is unclear how important this issue is. We have not investigated the issue but we did not encounter problems either. The issue is something that we believe could be worth investigating. It is likely that we have simply not thought of examples where Amigo would need to support inheritance. If Amigo is expanded to handle persistence of objects, and not just data retrieval, it will definitely become an issue.

As we have discussed there are parts of Amigo which will have to be implemented before it is useful to the programming community. The appeal or usefulness of programming languages is a very subjective matter. For some it is most important that it is easy to read and write, for others it is performance. Testing the language with programmers is very important and before Amigo could be released more tests have to be performed. It is especially interesting to know whether Amigo would be the first choice in all types of projects or only in certain types.

The focus of Amigo was to create an easy and intuitive query language. We have chosen not to focus on performance as it is something that can be done at a later stage. That is not to say that performance is not important, it is. But if no one will use the Amigo because they think it is not useful, it is not important how fast it performs. Enhancing the performance of Amigo involves among other things optimising the generated SQL queries. If the choice is to extend Amigo so it can handle returned data, an area of interest would be caching of results. In any case, testing the performance of Amigo as it is, would be the first thing to do.

The object of the Amigo project was to implement Amigo as a proof-of-concept, meaning determining if the core ideas are viable. We do not believe that the limitations have negatively affected this goal, quite the opposite. In the LIP project we aimed at creating a complete object relational mapper similar to Hibernate, on a smaller scale, and the result was that none of the separate parts were fully explored. As we discovered, object relation

mappers can be divided into virtually disconnected parts. We chose to focus on the query part and the limitations helped us keep the focus. If we look at what we have identified as missing in Amigo they are not critical areas in the sense that they might not be possible. For instance the fact that it is not possible to use `while` statements is not critical as we have shown that it is easily implemented as we have implemented `for` statements.

7.6 Tools

In the implementation and testing of the Amigo compiler a number of tools have been used. For the actual implementation we settled on .NET as execution platform, which in turn led to the choice of implementing the compiler in C#. We have been very satisfied with this choice. The main reason for this is, that the reflection API for generating native .NET intermediate language is relatively easy to use, although some problems during the implementation of the code generator did occur.

Generating the parser was easy, since the Coco/R parser generator posed no problems during implementation. Coco/R fit our needs very well, and it's syntax for the attributed grammar is quite easy to comprehend. Furthermore, it is very well documented compared to other parser generators we have worked with previously.

The integrated development environments (IDE) used in this project was Visual Studio.NET and also SharpDevelop on Microsoft Windows, and Monodevelop on Linux. Both the latter of these IDEs are open source, and are works-in-progress. Unfortunately, this means that they are both flawed in some circumstances, most notably Monodevelop. This has caused a bit of frustration during implementation, since work at times seemed wasted when dealing with the flaws of the IDE.

During testing we have made use of the NUnit framework. This has been of great help in determining errors in the compiler and the generated code.

As back-end data store, PostgreSQL was used. This DBMS is very feature rich, and we have been pleased working with this particular application.

Chapter 8

Future Work

In the course of this project, we have designed and implemented a new language for querying relational data from an object oriented setting. Although the implementation consists of a working compiler, there is still room for improvements and additional features in Amigo. It is clear that the limitations that were established for the Amigo project in the beginning of this report, needs to be addressed in the future. In this section, however, we will not get further into these, as they have already been thoroughly discussed. Instead, the focus is on the "big picture".

8.1 Language Integration

In Chapter 1, and throughout the report, we have stated that Amigo at one point should be integrated in some way or another in an object oriented host language. The main argument for integrating Amigo in a host language is to lower the coupling between application language and query language. This should result in writing both queries and general purpose code in a single language and provide a more natural feel for the programmer when developing applications in - for instance C# - and using Amigo as a query language for accessing a relational database.

The practical side of integrating Amigo in a host language poses a few different possible solutions. One solution could be to extend the host language's compiler, adding the syntax and semantics of Amigo. This approach is beneficial in the way, that the programmer only has to worry about using a single compiler, and that Amigo would exist as a natural part of the host language. However, our experience with integrating the LIP language features in the Mono C# compiler shows that this approach is quite difficult

in practice. The Mono C# compiler is a complex piece of software that is very specific in the task it performs. It takes a long time to get to know the source code in so much detail that it is possible to add several language features. Obviously this approach requires that the source code for the host language compiler is available. This severely limits the number of compilers for which Amigo could be integrated.

A more viable solution to the integration, could be to implement some kind of Amigo preprocessor for the host language. The implementation of such a preprocessor is relatively simple compared to the previously proposed solution. It is very likely that the currently implemented compiler could be reused in this approach - at least if the host language is available on the .NET platform. Furthermore, this approach would work for any object oriented host language - even languages where the compiler source code is not available.

8.2 Object-relational mapping

Amigo is only a query language and in this report the focus have been on developing the language and its features. No effort have been made concerning the actual returning of data from filters in strongly typed data structures. Multiple solutions exist that handles this, however. One approach is the one used in most object/relational mappers[5], where the mapped object is filled with the data using already built-in language constructs such as constructors or getters and setters. Another approach is seen in LINQ where a combination of type inference and anonymous types is used to return the requested data in a strongly typed anonymous type.

The actual mapping of the relational structure to objects in Amigo is done by reading mapping files from NHibernate. This is not the optimal solution. As we discussed when implementing LIP[16], many mapping features can be retrieved directly from the database if the database exposes its schema information. A hybrid of these approaches can be seen in the ActiveRecord project[1] which in fact is an abstraction built on top of NHibernate where many properties of the mapping is inferred. And in fact, the mapping can be generated using a tool which queries the database schema using OleDb.

The third solution is to integrate the mapping in the language itself in a manner so that persistent types are marked using special keywords. This is a large extension to the language, as properties such as length of fields in

the database would have to be specified on each persistent property if the database and the language does not share the same type system, which is rarely the case.

8.3 Industrial Application

It is our firm belief that the integration of Amigo in a host programming language is necessary if Amigo is to be accepted as a practical usable query language. The language itself may be intuitive and easy to use, but if the tools for developing software in the language are not up to par, most programmers will return to the technologies they are used to, thus abandoning Amigo entirely.

Making sure that the Amigo compiler, and the integration of Amigo in a host language works correctly and does not pose unnecessary problems for the programmer, requires a massive amount of testing. In this project we have only tested Amigo in a small setting. It would be very interesting to carry out an extensive and thorough test and evaluation of Amigo among professional software developers. This would allow us to make more qualified decisions of how to improve Amigo to facilitate use in an industrial setting.

Chapter **9**

Conclusion

In this report we have presented Amigo, a language for querying relational data from an object oriented setting. Amigo provides syntax and type checking of the queries at compile-time as opposed to traditional string based SQL queries that are not checked in anyway until runtime. Amigo was designed with the intention of creating an intuitive and concise query language.

The conclusions drawn from our experience in developing the LIP programming language provided a sound basis for further development of the filter concept initially conceived in the LIP project. A number of existing technologies and query languages were analysed before the design of Amigo began. These technologies served as inspiration for the development of the language, and some features specific to some of the analysed technologies found their way to the design of Amigo.

A proof-of-concept implementation of a compiler for the Amigo language has been realised in the course of this project as well. Implementing a working compiler has been of great importance to us, since we believe that a programming language needs to be tested in practice before any real conclusions can be drawn from it. Therefore a simple test of Amigo was performed among a few programmers in order for us to get an idea of how others saw Amigo. It is clear however, that this test does not suffice to give the full picture of the applicability and future potential of Amigo.

During the initial steps of this project period we quickly set up a number of goals for the project. We feel that most of these goals have been met to a high degree. Not necessarily in the way that we had thought, since as the project progressed we grew more knowledgeable in the area of query languages. Still, there are several issues that needs to be addressed in

order for Amigo to be complete and ready for practical use in a software development environment. These are issues that we are ready to delve deeper into at a later time, as we are confident that Amigo could in fact be a player in the field of statically typed query languages that seem to get more and more attention from the programming community at large.

Bibliography

[1] Castle Project - ActiveRecord.
http://www.castleproject.org/index.php/ActiveRecord.
[cited at p. 82]

[2] The CLI Specification.
http://msdn.microsoft.com/netframework/programming/clr/.
[cited at p. 57]

[3] HaskellDB Homepage at haskell.org.
http://www.haskell.org/haskellDB.
[cited at p. 17]

[4] HaskellDB Homepage at sourceforge.net.
http://haskelldb.sourceforge.net.
[cited at p. 17]

[5] Hibernate - relational persistence for idiomatic java.
http://www.hibernate.org/hib_docs/v3/reference/en/html/.
[cited at p. 19, 82]

[6] Microsoft's open database connectivity (odbc) interface.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/dasdkodbcoverview.asp.
[cited at p. 3]

[7] Official Coco/R Homepage.
http://www.ssw.uni-linz.ac.at/Coco/.
[cited at p. 48]

[8] Official Haskell Homepage.
http://www.haskell.org.
[cited at p. 17]

[9] Official Jay Homepage.
http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/.
[cited at p. 48]

[10] Official LINQ Homepage.

http://msdn.microsoft.com/data/ref/linq/.
[cited at p. 69]

[11] Official NUnit Homepage.
http://nunit.org/.
[cited at p. 62, 63]

[12] Official SableCC Homepage.
http://www.sablecc.org.
[cited at p. 48]

[13] PostgreSQL database manual.
http://www.postgresql.org/docs/manuals/.
[cited at p. 62]

[14] Suns's java database connectivity (jdbc) interface.
http://java.sun.com/products/jdbc/.
[cited at p. 3]

[15] Microsoft Corporation.
C# 3.0 language specification.
http://download.microsoft.com/download/9/5/0/9503e33e-fde6-4aed-b5d0-ffe749822f1b/csharp%203.0%20specification.doc.
[cited at p. 69]

[16] Rune Hammerskov, Jakob Andersen, and Lars Nielsen.
Language Integrated Persistence.
Report is unpublished but available through AAU.
[cited at p. 3, 8, 82]

[17] International Organization for Standardization.
ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL.
International Organization for Standardization, Geneva, Switzerland, 1992.
Available in English only.
[cited at p. 11]

[18] International Organization for Standardization.
ISO/IEC 9075-1:1999: Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework).
International Organization for Standardization, Geneva, Switzerland, 1999.
[cited at p. 11]

[19] Alex Krieger and Boris M. Trukhnov.
SQL Bible.
John Wiley & Sons, 2003.
[cited at p. 12]

[20] David Maier.
Representing database programs as objects.
In François Bancilhon and Peter Buneman, editors, *DBPL*, pages 377–386. ACM Press / Addison-Wesley, 1987.
[cited at p. 3]

[21] C. Robert Martin.
Agile Software Development, Principles, Patterns, and Practices.
Prentice Hall, 2002.
[cited at p. 48]

Appendices

Appendix A

Grammar

```
1  using e = Amigo.Elements;
2  using System.Collections;
3
4  COMPILER Amigo
5      /* Global variables n' stuff... */
6      private e.FilterCollection fc = new e.FilterCollection();
7
8      public e.FilterCollection Tree {
9             get { return fc; }
10     }
11
12     /* LL1 conflict resolver methods
13      *
14      */
15
16     /* identifier '=' */
17     bool IsAssignment() {
18         bool b = la.kind == _identifier && scanner.Peek().kind == _assign;
19         return b;
20     }
21
22     /* Type identifier '=' | Type identifier ';' */
23     bool IsVarDeclaration() {
24         Token x = scanner.Peek();
25         while (x.kind != _identifier) {
26             x = scanner.Peek();
27         }
28         Token y = scanner.Peek();
29         scanner.ResetPeek();
30         bool b = x.kind == _identifier && (y.kind == _assign || y.kind == _semicolon);
31         return b;
32     }
33
34
35 CHARACTERS
36     letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
37     digit       = "0123456789" .
38     cr          = '\r' .
39     lf          = '\n' .
40     tab         = '\t' .
```

```
41     nl          = cr + lf .
42     str         = ANY - '"' - '\\' - nl .
43     chr         = ANY - '\'' - '\\' - nl .
44
45 TOKENS
46     identifier  = letter { letter | digit } .
47     number      = digit { digit } .
48     tstr        = '"' { str } '"' .
49     tchr        = '\'' { chr } '\'' .
50
51     assign      = '=' .
52     semicolon   = ';' .
53     colon       = ':' .
54     comma       = ',' .
55     dot         = '.' .
56     neg         = '!' .
57     lcurly      = '{' .
58     rcurly      = '}' .
59     lparen      = '(' .
60     rparen      = ')' .
61     lbrack      = '[' .
62     rbrack      = ']' .
63
64     plus        = '+' .
65     minus       = '-' .
66     div         = '/' .
67     mul         = '*' .
68     eq          = "==" .
69     neq         = "!=" .
70     lt          = '<' .
71     gt          = '>' .
72     lteq        = "<=" .
73     gteq        = ">=" .
74     ijoin       = "===" .
75     lijoin      = "|=" .
76     rijoin      = "=|" .
77     lojoin      = "|~" .
78     rojoin      = "~|" .
79     crjoin      = "><" .
80     trjoin      = "->" .
81
82     if          = "if" .
83     else        = "else" .
84     var         = "var" .
85     for         = "for" .
86     or          = "or" .
87     and         = "and" .
88     public      = "public" .
89     private     = "private" .
90     init        = "init" .
91     object      = "object" .
92     value       = "value" .
93     where       = "where" .
94     join        = "join" .
95     group       = "group" .
96     order       = "order" .
97     asc         = "ASC" .
98     desc        = "DESC" .
99     string      = "string" .
```

```
100     int         = "int" .
101     float       = "float" .
102     bool        = "bool" .
103     char        = "char" .
104     null        = "null" .
105     true        = "true" .
106     false       = "false" .
107
108 COMMENTS FROM "/*" TO "*/" NESTED
109 COMMENTS FROM "//" TO cr lf
110
111 IGNORE cr + lf + tab
112
113 PRODUCTIONS
114     Amigo                      (. e.Filter f; .)
115     =
116         { Filter<out f>        (. fc.Filters.Add(f); .)
117         }
118         .
119
120     Filter<out e.Filter f>     (. string name; string mod; e.FilterBlock fb;
121                                   e.VariableList pl; .)
122     =
123         Modifier<out mod>
124         var
125         Identifier<out name>   (. f = new e.Filter(name, mod); .)
126         lparen
127         [ ParamList<out pl>    (. f.Parameters = pl; .)
128         ]
129         rparen
130         lcurly
131         { Block<out fb>        (. f.Blocks.Add(fb); .)
132         }
133         rcurly
134         .
135
136     ParamList<out e.VariableList pl>(. e.Variable v = null; pl = new e.VariableList();
137                                   string vt = null; bool ita = false; .)
138     =
139         Type<out vt, out ita>
140         Variable<out v>        (. v.Type = vt; v.IsArrayType = ita; pl.Add(v); .)
141         { comma
142           Type<out vt, out ita>
143           Variable<out v>      (. v.Type = vt; v.IsArrayType = ita; pl.Add(v); .)
144         }
145         .
146
147     Type<out string vt, out bool ita>(. string type = null; ita = false; vt = null; .)
148     =
149         ( SimpleType<out type>  (. vt = type; .)
150         | Identifier<out type>  (. vt = type; .)
151         )
152         [ lbrack
153           rbrack               (. ita = true; .)
154         ]
155         .
156
157     ArgList<out e.ExpressionList al>(. al = new e.ExpressionList(); e.Expression exp = null; .)
158     =
```

```
159         Expression<out exp>        (. al.Add(exp); .)
160         { comma
161           Expression<out exp>      (. al.Add(exp); .)
162         }
163         .
164
165     Variable<out e.Variable v>     (. string s = null, n = null, name; e.Expression exp; .)
166     =
167         Identifier<out name>       (. s += name; .)
168         { dot                      (. s += "."; .)
169           Identifier<out name>     (. s += name; .)
170         }                          (. if (s.Contains(".")) {
171                                         n = s.Substring(0, s.LastIndexOf('.'));
172                                         s = s.Substring(s.LastIndexOf('.')+1);
173                                       }
174                                       v = new e.Variable(s, n); .)
175         [ lbrack
176           Expression<out exp>      (. v.Index = exp; .)
177           rbrack
178         ]
179         .
180
181     VarDeclaration<out e.VarDeclaration vd>(. e.Variable v; string vt = null; bool ita = false;
182                                               e.Expression exp; .)
183     =
184         Type<out vt, out ita>
185         Variable<out v>            (. v.Type = vt; v.IsArrayType = ita;
186                                       vd = new e.VarDeclaration(v); .)
187         [ assign
188           ( Expression<out exp>    (. vd.Value = exp; .)
189             | lcurly
190               Expression<out exp>  (. vd.ValueSet.Add(exp); .)
191               { comma
192                 Expression<out exp> (. vd.ValueSet.Add(exp); .)
193               }
194               rcurly
195           )
196         ]
197         .
198
199     Assignment<out e.Assignment a>   (. e.Variable v; e.Expression exp; .)
200     =
201         Variable<out v>
202         assign
203         Expression<out exp>          (. a = new e.Assignment(v, exp); .)
204         .
205
206     Limit<out int? lf, out int? lt>  (. lf = null; lt = null; int n;.)
207     =
208         [ Number<out n>              (. lf = n; .)
209         ]
210         dot dot
211         [ Number<out n>              (. lt = n; .)
212         ]
213         .
214
215
216     /** Block productions **/
217     Block<out e.FilterBlock fb>      (. fb = null; e.InitBlock ib; e.ValueBlock vb;
```

```
218                                    e.JoinBlock jb; e.WhereBlock wb; e.GroupBlock gb;
219                                    e.OrderBlock ob; e.ObjectBlock obj; .)
220     =
221         InitBlock<out ib>           (. fb = ib; .)
222         | ObjectBlock<out obj>      (. fb = obj; .)
223         | ValueBlock<out vb>        (. fb = vb; .)
224         | JoinBlock<out jb>         (. fb = jb; .)
225         | WhereBlock<out wb>        (. fb = wb; .)
226         | GroupBlock<out gb>        (. fb = gb; .)
227         | OrderBlock<out ob>        (. fb = ob; .)
228         .
229
230     InitBlock<out e.InitBlock ib>   (. e.VarDeclaration vd; e.Statement stmt = null;.)
231     =
232         init                        (. ib = new e.InitBlock(); .)
233         colon
234         { ( IF (IsVarDeclaration())
235             VarDeclaration<out vd>  (. ib.VarDeclarations.Add(vd); .)
236             semicolon
237             | Statement<out stmt>   (. ib.Statements.Add(stmt); .)
238           )
239         }
240         .
241
242     ObjectBlock<out e.ObjectBlock obj>(. e.Variable name; int? lf = null; int? lt = null; .)
243     =
244         object
245         colon
246         [ lbrack
247           Limit<out lf, out lt>
248           rbrack
249         ]
250         Variable<out name>          (. obj = new e.ObjectBlock(name, lf, lt); .)
251         semicolon
252         .
253
254     ValueBlock<out e.ValueBlock vb>  (. e.Statement stmt = null; .)
255     =
256         value                       (. vb = new e.ValueBlock(); .)
257         colon
258         { Statement<out stmt>       (. vb.Statements.Add(stmt); .)
259         }
260         .
261
262     JoinBlock<out e.JoinBlock jb>   (. e.Statement stmt = null; .)
263     =
264         join                        (. jb = new e.JoinBlock(); .)
265         colon
266         { Statement<out stmt>       (. jb.Statements.Add(stmt); .)
267         }
268         .
269
270     WhereBlock<out e.WhereBlock wb>  (. e.Statement stmt = null; .)
271     =
272         where                       (. wb = new e.WhereBlock(); .)
273         colon
274         { Statement<out stmt>       (. wb.Statements.Add(stmt); .)
275         }
276         .
```

```
277
278     GroupBlock<out e.GroupBlock gb>  (. e.Statement stmt = null; .)
279     =
280         group                        (. gb = new e.GroupBlock(); .)
281         colon
282         { Statement<out stmt>        (. gb.Statements.Add(stmt); .)
283         }
284         .
285
286     OrderBlock<out e.OrderBlock ob>  (. e.Statement stmt = null; .)
287     =
288         order                        (. ob = new e.OrderBlock(); .)
289         colon
290         { Statement<out stmt>        (. ob.Statements.Add(stmt); .)
291         }
292         .
293
294
295     /** Statement productions **/
296
297     Statement<out e.Statement stmt>  (. e.Assignment a; e.OrStatement ors; e.AndStatement ands;
298                                         e.OrderExpression oe; e.Expression exp; e.BlockStatement bs;
299                                         e.ForStatement fors; stmt = null; e.IfElseStatement ies; .)
300     =
301             BlockStatement<out bs>       (. stmt = bs; .)
302         | ForStatement<out fors>     (. stmt = fors; .)
303         | IfElseStatement<out ies>   (. stmt = ies; .)
304         | OrStatement<out ors>       (. stmt = ors; .)
305         | AndStatement<out ands>     (. stmt = ands; .)
306         | OrderExpression<out oe>    (. stmt = oe; .)
307         | ( IF (IsAssignment()) Assignment<out a>(. stmt = a; .)
308             semicolon
309             |
310             Expression<out exp>      (. stmt = exp; .)
311             semicolon
312           )
313         .
314
315     BlockStatement<out e.BlockStatement bs>(. bs = new e.BlockStatement(); e.Statement st = null; .)
316     =
317             lcurly
318             { Statement<out st>          (. bs.Contents.Add(st); .)
319             }
320             rcurly
321             .
322
323     ForStatement<out e.ForStatement stmt>(. e.VarDeclaration vd; e.Assignment a;
324                                             e.Expression exp; e.Statement st; .)
325     =
326         for                          (. stmt = new e.ForStatement(); .)
327         lparen
328         VarDeclaration<out vd>       (. stmt.VarDeclaration = vd; .)
329         semicolon
330         Expression<out exp>          (. stmt.Expression = exp; .)
331         semicolon
332         Assignment<out a>            (. stmt.Assignment = a; .)
333         rparen
334         Statement<out st>            (. stmt.Statement = st; .)
335         .
```

```
336
337    IfElseStatement<out e.IfElseStatement stmt>(. stmt = new e.IfElseStatement(); e.Expression exp;
338                                              e.Statement st = null; .)
339    =
340            if
341            lparen
342            Expression<out exp>         (. stmt.IfExpression = exp; .)
343            rparen
344            Statement<out st>           (. stmt.IfStatement = st; .)
345            [ else
346              Statement<out st>         (. stmt.ElseStatement = st; .)
347            ]
348            .
349
350    OrStatement<out e.OrStatement stmt>(. stmt = new e.OrStatement(); e.Statement st; .)
351    =
352        or
353        Statement<out st>           (. stmt.Statement = st; .)
354        .
355
356    AndStatement<out e.AndStatement stmt>(. stmt = new e.AndStatement(); e.Statement st; .)
357    =
358        and
359        Statement<out st>           (. stmt.Statement = st; .)
360        .
361
362
363    /** Expression productions **/
364
365    OrderExpression<out e.OrderExpression oe>(. e.Variable v; bool asc = false; oe = null; .)
366    =
367        lbrack
368        asc                         (. asc = true; .)
369        | desc
370        rbrack
371        Variable<out v>             (. oe = new e.OrderExpression(v, asc); .)
372        semicolon
373        .
374
375    Expression<out e.Expression exp> (. exp = null; e.Expression rel = null;
376                                        e.Expression rer = null; string op = null;
377                                        e.BinaryExpression bin = null; e.Expression temp = null; .)
378    =
379        RelExpression<out rel>      (. temp = rel; .)
380        [ JoinOperator<out op>      (. bin = new e.BinaryExpression(); bin.Left = temp;
381                                        bin.Operator = op; .)
382          RelExpression<out rer>    (. bin.Right = rer; temp = bin; .)
383        ]                           (. exp = temp; .)
384        .
385
386    RelExpression<out e.Expression exp>(. exp = null; e.Expression sel = null;
387                                          e.Expression ser = null; string op = null;
388                                          e.BinaryExpression bin = null; e.Expression temp = null; .)
389    =
390        SimpleExpression<out sel>   (. temp = sel; .)
391        [ RelOperator<out op>       (. bin = new e.BinaryExpression(); bin.Left = temp;
392                                        bin.Operator = op; .)
393          SimpleExpression<out ser> (. bin.Right = ser; temp = bin; .)
394        ]                           (. exp = temp; .)
```

```
395         .
396
397
398     SimpleExpression<out e.Expression exp>(. exp = null; string op = null;
399                                             e.Expression tel = null; e.Expression ter = null;
400                                             e.Expression temp = null; e.BinaryExpression bin = null; .)
401     =
402         Term<out tel>               (. temp = tel; .)
403         { AddOperator<out op>       (. bin = new e.BinaryExpression(); bin.Left = temp;
404                                        bin.Operator = op; .)
405           Term<out ter>             (. bin.Right = ter; temp = bin; .)
406         }                           (. exp = temp; .)
407         .
408
409     Term<out e.Expression exp>      (. string op = null; exp = null; e.Expression facl;
410                                        e.Expression facr; e.Expression temp; e.BinaryExpression bin; .)
411     =
412         Factor<out facl>            (. temp = facl; .)
413         { MulOperator<out op>       (. bin = new e.BinaryExpression(); bin.Left = temp;
414                                        bin.Operator = op; .)
415           Factor<out facr>          (. bin.Right = facr; temp = bin; .)
416         }                           (. exp = temp; .)
417         .
418
419     MemberAccessor<out e.MemberAccessor ma>(. ma = new e.MemberAccessor(); e.Variable v;
420                                              e.ExpressionList al; .)
421     =
422         Variable<out v>             (. ma.Variable = v; .)
423         [ lparen                    (. ma.IsProcedure = true; .)
424           [ ArgList<out al>         (. ma.Arguments = al; .)
425           ]
426           rparen
427         ]
428         .
429
430     Factor<out e.Expression fac>    (. fac = null; int number; string s; bool tf; char c;
431                                        e.Expression exp; bool negated = false; e.MemberAccessor ma; .)
432     =
433         [ neg                       (. negated = true; .)
434         ]
435         lparen
436         Expression<out exp>         (. exp.Negated = negated; fac = exp; .)
437         rparen
438         | MemberAccessor<out ma>    (. fac = ma; .)
439         | Number<out number>        (. fac = new e.NativeType("int", number); .)
440         | String<out s>                 (. fac = new e.NativeType("string", s); .)
441         | Null                      (. fac = new e.NullType(); .)
442         | TrueOrFalse<out tf>       (. fac = new e.NativeType("bool", tf); .)
443         | Char<out c>               (. fac = new e.NativeType("char", c); .)
444         .
445
446
447     /** Operator productions **/
448     JoinOperator<out string op>     (. op = null; .)
449     =
450                 ijoin               (. op = t.val; .)
451                 | lijoin            (. op = t.val; .)
452                 | rijoin            (. op = t.val; .)
453                 | lojoin            (. op = t.val; .)
```

```
454                 | rojoin              (. op = t.val; .)
455                 | crjoin              (. op = t.val; .)
456                 | trjoin              (. op = t.val; .)
457                 .
458
459     RelOperator<out string op>        (. op = null; .)
460     =
461                 eq                    (. op = t.val; .)
462                 | neq                 (. op = t.val; .)
463                 | lt                  (. op = t.val; .)
464                 | gt                  (. op = t.val; .)
465                 | lteq                (. op = t.val; .)
466                 | gteq                (. op = t.val; .)
467                 .
468
469
470     AddOperator<out string op>        (. op = null; .)
471     =
472                 plus                  (. op = t.val; .)
473                 | minus               (. op = t.val; .)
474                 .
475
476     MulOperator<out string op>        (. op = null; .)
477     =
478                 mul                   (. op = t.val; .)
479                 | div                 (. op = t.val; .)
480                 .
481
482
483     /** Simple productions **/
484     Modifier<out string mod>          (. mod = null; .)
485     =
486         private                       (. mod = t.val; .)
487         | public                      (. mod = t.val; .)
488         .
489
490     SimpleType<out string st>         (. st = null; .)
491     =
492         int                           (. st = t.val; .)
493         | float                       (. st = t.val; .)
494         | string                      (. st = t.val; .)
495         | char                        (. st = t.val; .)
496         | bool                        (. st = t.val; .)
497         .
498
499     TrueOrFalse<out bool b>           (. b = false; .)
500     =
501         true                          (. b = true; .)
502         | false                       (. b = false; .)
503         .
504
505     Number<out int number>
506     =
507         number                        (. number = Convert.ToInt32(t.val); .) .
508
509     Null
510     =
511         null .
512
```

```
513     String<out string s>
514     =
515         tstr                        (. s = t.val.Replace("\"", ""); .) .
516
517     Char<out char c>
518     =
519         tchr                        (. c = Convert.ToChar(t.val.Replace("'", "")); .) .
520
521     Identifier<out string name>
522     =
523         identifier                  (. name = t.val; .) .
524
525 END Amigo .
```

Appendix B

Amigo manual

The Amigo query language is an object oriented query language. Amigo works by translating Amigo source code into SQL queries. Amigo's core idea is a method-like construct called a filter. Filters are built up of filter-blocks. Filters cannot be defined within other filters but existing filters can be called from within other filters; see explanation of the `init` block.

Before we describe the different filter blocks we will look at a few general issues.

First, all mapping between the database and the application is defined in Hibernate mapping files which you can read more about on the Hibernate website at *www.hibernate.org*.

Second, you can declare variables, use `for` statements and `if-else` statements in most of the filter-blocks. The syntax and semantics of these are the same as in most general purpose languages. Declaration of variables in the `init` block works a little different; see explanation of the `init` block.

Arithmetic operations

Amigo support the following arithmetic operations using infix notation:

- $*$ Multiplication
- $/$ Division
- $+$ Addition
- $-$ Subtraction

The arithmetic operator could for instance be use in the `init` block like this:

```
public var Arithmetic() {
    init:
        int i = 5 + 2 * 10;
}
```

The variable `i` is assigned the value 25. The basic types of the Amigo language are: *int*, *float*, *bool*, *char*, and *string*. Strings are specified in `".."`.

Boolean expressions

Boolean expression can be constructed using comparison operators to get boolean values from comparing values and variables. `True` and `False` values can be used.

The comparison operators supported by Amigo are:

- !=
- ==
- <=
- >=
- <
- >

Filter Blocks

There are seven types of filter-blocks:

- object
- init
- value
- join
- where
- group
- order

The object block

In the `object` block you can define which kind of objects the filter should return. If a condition is added in the `where` block comparing a single value to a primary or unique column in the database the result is a single value otherwise the result is a collection. If a `value` block is present the filter will return a single value.

In this example we have a filter called `AverageSalary` which take no arguments.

```
public var AverageSalary() {
    object:
        Employee;
}
```

The `object` block contains one return type, `Employee`. This example would correspond to `SELECT * FROM Employees` in SQL.

It is possible to limit the returned result to only a certain number. This is a feature which works exactly like it does in SQL. Amigo allows you to limit in three ways

- $[x..y]$ Limits from index x to index y
- $[..y]$ Limits to index y
- $[x..]$ Limits from index x

You use a prefix notation like this

```
public var AverageSalary() {
    object:
        [..20]Employee;
}
```

if you only want the first 20 results.

The value block

In the `value` block you are able to define a return result other than objects or collections of objects. This is primarily used for aggregate functions, as any method call in this block is matched to a function in the database. If the method call cannot be mapped to a function an error occurs.

In this example we have a filter called `AverageSalary`.

```
public var AverageSalary() {
    object:
        Employee;
    value:
        AVG(Employee.Salary);
}
```

Just as the previous example, the `object` block contains one return type, `Employee`. But now we have added a `value` block containing a method call to `AVG` which can be mapped to an aggregate function of the same name in the database. Adding the `value` block means the filter returns a single value, because the `AVG` returns a single value, namely the average salary.

The init block

In the `init` block you can define variables which can be used in the entire filter. Variables can contain any type of data including results from other filters. This is the only way to use filters within filters.

In this example we have a filter called `Employees` which takes an integer as argument. We also have an `init` block a variable declarations.

```
public var Employees(int age) {
    init:
        float X = AverageSalary(age);
}
```

`X` is assigned the value of another filter called `AverageSalary`. This is the filter we created in the previous two examples. `AverageSalary` takes an integer `age` as argument is the argument the `Employees` filter takes.

The where block

In the `where` you can specify how returned data should be filtered. You do this by using boolean expressions. These expressions work on properties on

objects which can be compared with simple types or values defined in the `init` block or the filter parameters.

In this example, we extend the `AverageSalary` filter with a `where` block.

```
public var AverageSalary(int age) {
    object:
        Employee;
    value:
        AVG(Employee.Salary);
    where:
        Employee.Age >= age;
}
```

The `where` block contains a boolean expression which makes the filter return the average salary for employees over a certain age.

The `where` block can contain any number of boolean expressions. Amigo will interpret all boolean expressions in the `where` block as if there were an `AND` in between. That is, if you write:

```
public var AverageSalary(int age) {
    object:
        Employee;
    value:
        AVG(Employee.Salary);
    where:
        Employee.Age >= age;
        Employee.Salary > 20000;
}
```

Amigo will interpret this as employees who's age is more than `age` `AND` who's salary is more than 20000. If you want employees who's age is more than `age` `OR` who's salary is more than 20000 you need to use an `OR` block like this:

```
public var AverageSalary(int age) {
    object:
        Employee;
    value:
        AVG(Employee.Salary);
    where:
        or {
            Employee.Age >= age;
            Employee.Salary > 20000;
        }
}
```

You can have additional expressions before or after the `or` block and Amigo will `and` them.

The join block

The `join` block is used to give you the option of joining tables as you would in SQL. Amigo supports the same types of joins as SQL. In addition to the join types present in SQL, Amigo also introduce the tree- join operator which performs a recursive join on tables representing a tree structure in the database. This operator will have the column that points to the parent on the right side and the column that points to the child column on the right and will retrieve all nodes in a branch according to conditions set in the where block. Here is an example of how to write a tree join:

```
public var EmployeesByDepartmentName(string name) {
    object:
        Employee;
    join:
        Employee.DepartmentID === Department.DepartmentID;
    where:
        Department.Name == name;
}
```

The left/right side of the join operators (except tree-join) can be a primary-foreign key relationship or any other two fields the user might want to join. Amigo allows just for two object names to be on both sides and their primary- foreign key relationship in the database will be inferred from the mapping files if possible.

As you can see in the example Amigo uses an infix notation to indicate the type of join. All the join types are listed here along with the corresponding operator:

- `===` Inner join
- `|=` Left-Inner join
- `=|` Right-Inner join
- `|~` Left-Outer join
- `~|` Right-Outer join
- `><` Cross join
- `->` Tree join

The order block

The order block is used to sort the returned data either ascending or descending according to one or more properties. We use prefix notation with either `[ASC]` or `[DESC]`.

In this example, we want all employees ordered after their salary beginning with highest paid

```
public var OrderedEmployees() {
    object:
        Employee;
    order:
        [DESC]Employee.Salary;

}
```

The group block

The `group` block allows you to group results. Either by a specific property or even to return an aggregate value based on grouping.

In this example we expand the previous example by grouping the employees by department

```
public var OrderedEmployees() {
    object:
        Employee;
    value:
        SUM(Department.Salary);
    group:
        Employee.DepartmentID;
}
```

giving us all employees ordered by salary and grouped by department.

Conditional statements

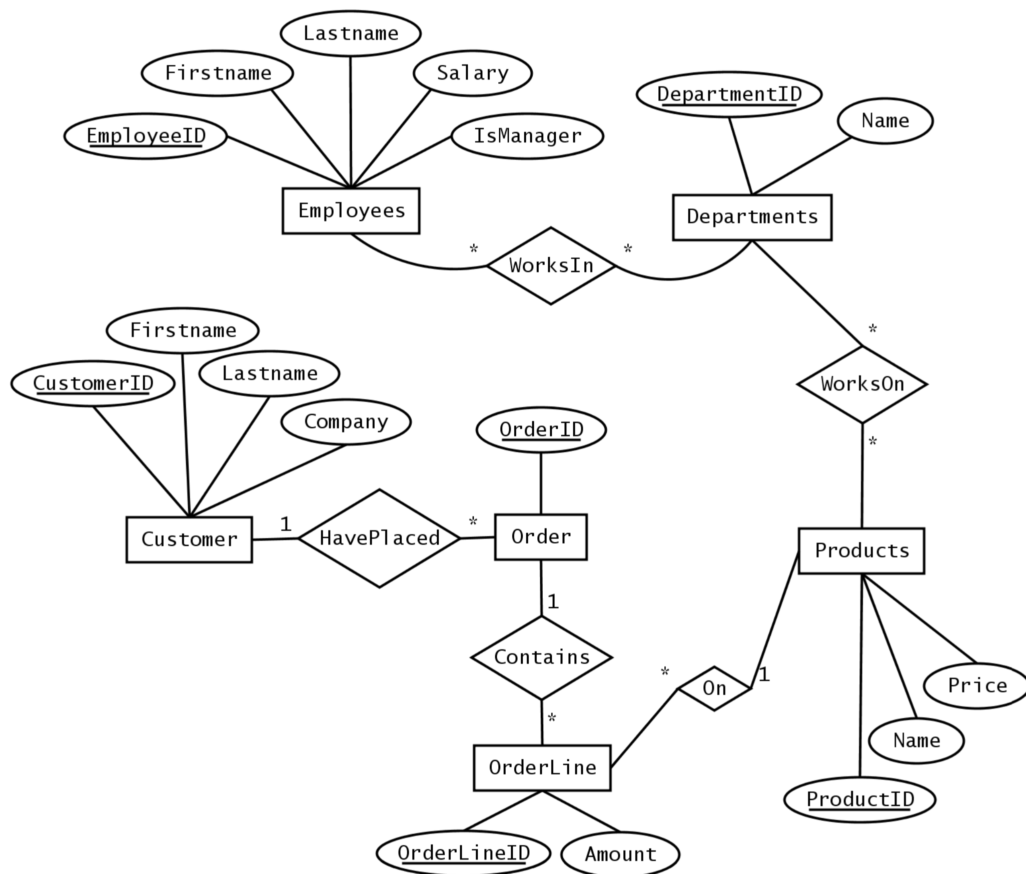
Amigo supports `for` statements and `if-else` statements in any block in the filters. They work like they do in Java and C#, so a filter using an `if-else` statement could look like this:

```
public var (int modifier) {
    object:
        Employee;
    where:
        if(modifier == 1) {
            Employee.Salary <= 10000;
        } else if (modifier == 2) {
            Employee.Salary <= 10000;
            Employee.IsManager == true;
        } else {
            Employee.Salary <= 10000;
            Employee.IsManager == false;
        }
}
```

Appendix C

Validation test

In this test you will be presented with nine exercises. Each exercise describes a data retrieval problem. It is formulated in spoken English and you are asked to write Amigo code that solves the problem. All the exercises assumes that a database already exists. The database is visualised in the following ER-diagram:



An exercise could look like this:

"Write one or more filters which will retrieve all employees with a salary of more than 20.000."

and the solution to this exercise could look something like this:

```
public var HighPaidEmployees() {
    object:
        Employee;
    where:
        Employee.Salary > 20000;
}
```

Additionally you are asked to evaluate how easy or difficult it was to do the exercise.

After completing all the exercises you are asked to write a short evaluation of Amigo.

Exercise 1

"Write one or more filters which will return all departments."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Exercise 2

"Write one or more filters which will return the 3 lowest paid employees."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Exercise 3

"Write one or more filters which will return the number of employees who are managers."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Exercise 4

"Write one or more filters which will return all orders made by the customer with ID 3."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Exercise 5

"Modify the filter, or filters, from exercise 4 so that it will return all orders made by the customer from the company "Smittys Shoeshine Inc."."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Exercise 6

"Write one or more filters which will return all employees with salaries under 10.000 or above 20.000."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Exercise 7

"Write one or more filters which will return the average salary for each department."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Exercise 8

"Write one or more filters which can take an integer as argument. If the integer is 1, the filter must return all employees with salaries below 10.000. If the integer is 2 the filter must return all employees with a salaries between 10.000 and 20.000. If the integer is 3 the filter must return all employees with salaries above 20.000."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Exercise 9

"Write one or more filters which will return all employees working on the product with ID 2. The employees should be ordered by department."

Please indicate how difficult you found this exercise

(Easy) 1 2 3 4 5 (Difficult)

Please write a short evaluation of Amigo. What are the strong points and what are the week points. If there are things which you think should be changed, what are they?

Appendix **D**

Summary

The general problem that is dealt with in this report, is that of weakly typed explicit queries. In object oriented languages a relational database is accessed through Call Level Interfaces (CLI) using SQL embedded in strings. These strings are neither syntax nor type checked. This imposes a problem for programmers in the fact that development time is extended or that errors are not caught before the application is shipped to the end-user.

This report is based on a former report called Language Integrated Persistence (LIP), which also dealt with this problem. In LIP a method-like construct called a filter was proposed. In this report the filter concept is developed further and extended to make better use of native Relational Database Management System (RDBMS) features. The extended concept of filters is the basis for the query language Amigo. This report describes the design of the language and an implementation of a proof-of-concept compiler for the language.

Several technologies, which deal with the same issue as Amigo, are analysed to determine what features Amigo should include. The technologies in question are: SQL, PL/SQL, T-SQL, HaskellDB, and Hibernate Query Language (HQL). The result of the analysis is that Amigo has to be statically checked and strongly typed and bridge the object oriented and the relational worlds in a manner that serves both. Inner, left, right, and outer join are supported, as well as standard aggregate function like `MAX`, `MIN`, `AVG`, and `COUNT`. Combination of filters is also supported to provide functionality that is similar to SQL's subselects.

Designing Amigo required a number of limitations to help keep focus on designing a safe query language that serves both the object oriented world as well as the relational. The functionality of Amigo is presented in

a series of examples. The main concept of Amigo is filters and blocks. A filter is made up of filter blocks. There are seven blocks: `init`, `object`, `where`, `value`, `join`, `order`, and `group` block. Additionally Amigo supports general purpose construct such as `if-else` and `for` statements. The full syntax is given in EBNF form and the semantics explained in a structured but informal manner. The type system, and especially the type inference mechanisms are presented along with a description of how the mapping between the database and Amigo is handled.

The implementation of the Amigo compiler is written in C# and the process is described in the report.

Amigo is tested using traditional testing methods. Unit tests are performed to test the correctness of the compiler. Also a validation test involving two test subjects is carried out. The main result of the test is that Amigo is intuitive and easy to use. A part of the language syntax might need to be altered slightly, but the test subjects did not agree with each other on this particular area.

Two related technologies are compared to Amigo in the report. First Microsoft's LINQ project and next Hibernate and in particular Hibernate's query language HQL.

Following this, all parts of the report is evaluated along with an evaluation of Amigo itself. The test with two subjects is evaluated as being a successful test, but the time each subject had was not enough. One of the subjects pointed out, that a test using a more complex scenario could be out. Following the evaluation is a description of which areas could be of interest to work on in the future. Basically, further testing is required to determine what has to be modified or added to Amigo if it is to be a competitor to related technologies.

Finally a conclusion is reached where the main point is that the project has been a success. There is still work ahead before Amigo will be able to compete with similar projects, but the main concepts of Amigo do not need to be changed.

List of Figures

3.1 Diagram of example scenario 26

4.1 Class diagram for the Elements library 51
4.2 A subtree in the concrete syntax tree 53
4.3 Structure of builders 57

Listings

1 Test vi
2.1 LIP example 9
2.2 LIP example 9
3.1 Amigo EBNF grammar 34
3.2 NHibernate 43
3.3 NHibernate 44
4.1 Filter element 50
4.2 Relationship mapping 56
4.3 Buidling an assembly using TypeBuilder, ModuleBuilder etc. 58