Functional Programming in a Basic Database Course[1]

Pieter Koopman, Vincent Zweije

Computer Science, Leiden University,

Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands

email: pieter@wi.leidenuniv.nl

Abstract

This paper describes why and how a functional programming language was used in an introductory database course. The purpose of the programming exercises in this course is to give students a better understanding of the internal structure and use of databases and database management systems.
We used a functional language for its high level of abstraction and the automatic memory management which make writing a simple database management system considerably easier.
Although the students had no previous knowledge of functional programming, they were capable to obtain useful experience in the database field. In order to enable students to concentrate on the database aspects of the exercises and to make rather elaborated systems in a limited amount of time, we supplied skeletons of the programs to make. Only the parts that are the core of the exercise had to be written by the students.
The exercises appear to serve their purpose very well. The corresponding parts of the exams are made considerably better since the introduction of these exercises in the course. After some initial hesitation, the students indicate that they prefer a functional language for these exercises above the imperative languages they know.

1. Introduction

This paper describes how functional programming is used in an elementary database course and the experiences with this use. The database course is situated in the second year of the computer science curriculum of four years for university students. The goal of the course is to make students aware of the reasons of existence for database management systems and to give a firm introduction to relational databases. We treat the design of relational schemas including normal forms and the query languages relational calculus, relational algebra and SQL. Also the more old-fashioned hierarchical

[1]This paper occurs also as: P. Koopman and V. Zweije: *Functional programming in a basic database course*. In: P. H. Hartel and M. J. Plasmeijer (editors): *Functional programming languages in education FPLE*, Nijmegen, The Netherlands, Dec 1995, LNCS 1022, Springer-Verlag, Heidelberg, pp 215-230.

and network model are discussed briefly. See also appendix A for additional course description.

Programming exercises are used to make students familiar with the construction and use of ad hoc databases and simple database management systems. The reason for having programming exercises instead of using existing database systems is that it is an important goal of this course to given students a clear view of the background and internals of database management systems, this is explained in detail in section 2.

The students have one year experience in imperative programming (using Pascal and C) and no experience with functional programming. The high level of abstraction and the automatic memory management are the reasons to use a functional language in this course. Especially the excellent abilities to manipulate lists can be used extremely well for a simple implementation of relations. See also section 3. Details of the organisation and exercises are given in section 4 and 5. Section 6 gives some reactions of the students. Finally, there is a discussion in section 7.

2. Role of Programming in the Course

In previous instances of this basic database course the students did practical work with an existing relational database management system (INGRES). The exercises consisted of writing some queries on an existing database and changing the contents and structure of this database using SQL-commands.

Although these exercises taught the students to work in SQL, we were not satisfied with the skills and insight of the students. In particular the understanding of what is happening when an SQL-statement is executed was low. As a consequence they could not predict the cost of a given manipulation. Also the rationale behind many design decisions remained misty. Using a database system teaches students about the internals of the systems what the students of a Pascal course learn about compiler construction: basically nothing. These exercises also did not teach the students anything about writing queries in relational calculus or relational algebra. Neither is writing SQL-statements helpful for understanding the other data models treated in the database course.

To solve these problems we replaced the practical work with an existing relational database management system by exercises in which the students build a simple database management system (DBMS) themselves and use this system to manipulate some data. Other programming exercises broaden the field of topics covered by the practical work. The total amount of practical work for the students is increased by this change.

Although many important topics of the database course are covered by the programming exercises, there are also additional exercises for the student about the other issues of the course. Later on in the curriculum, the students can learn how a state of the art relational DBMS should be used.

3. Why Functional Programming

Once we had decided to replace the exercises with the relational DBMS with the implementation of some DBMSes we had to select a suitable programming language. The two obvious candidates are the imperative languages familiar to the students: Turbo Pascal and C.

Especially the relational DBMS to construct requires the extensive use and dynamic creation of tables (relations). We want to prevent that the memory management involved attracts too much attention from the students. One option is to supply a package to store and manipulate relations in one of the imperative programming languages.

The lists which are standard available in functional programming languages are an excellent implementation of the relations used in our DBMS. In fact the list comprehensions in functional languages and the relational calculus share the same mathematical basis: Zermelo-Fraenkel set theory [Fraenkel 1922, Zermelo 1908]. No matter how sophisticated the relational package supplied with an imperative language is, it will be less usable and its syntax will always be inferior to the possibilities in a functional language. Together with the well known advantages of functional programming languages (they enable the construction of compact and understandable programs at a high level of abstraction that can be written fast) this is the reason to use a functional programming language in this course.

The Chosen Functional Language

After the decision to use a functional language we had to choose which language we were going to use. As indicated before, the students have no previous knowledge of functional programming. This means that we had free choice. However, since this is not a course in functional programming, a very simple and easy to explain yet powerful language is required. This makes an interpreter more suited than a compiler. Due to the extensive list manipulations that will be necessary the availability of list comprehensions (ZF-expressions) is a prerequisite. Speed is not considered to be of prime importance. Fancy type systems and other extensions are not required, nor wanted (they attract unnecessary attention).

Based on these requirements and the availability at our institute we have chosen Miranda[2].[Turner 85] Another good candidate was Gofer [Jones 94]. This language has a more powerful and hence more complex type system. An advantage of Gofer was its better availability, especially for students working at home on a PC. The reasons to select Miranda are the straightforward type system and simple, but sufficiently powerful, IO mechanism. Although we are satisfied with this choice, other functional languages can be used as an alternative.

[2]Miranda is a trade mark of Research Software Ltd.

4. Organisation of the Practical Work

The students are supposed to work in total four weeks full-time (about 160 hours) on this course. This time is spread over the semester of thirteen weeks. Each week there are two lecture hours. These lectures cover parts I, II, III and IV of the textbook of Elmasri and Navathe [Elmasri 94]. In addition there is a session of two hours were students can work on all exercises of this course (both the programming exercises and the pen and paper exercises) under supervision and with direct support. The remaining time should be spent on studying the topics covered in the lectures, implementation of the exercises and making the other exercises. Students are expected to spend about 40 hours in total to each of these three parts. As documentation for functional programming we supply copies of overhead sheets and a copy of the paper [Turner 85]. The Miranda system has an on-line manual.

The primary goal of the practical work is not to teach students how functional programs must be constructed, but to teach them database topics. In order to enable the novice functional programmers to construct useful database programs without spending much time on problems with functional programming, we give them much support. This support consists of relevant examples and a partial solution of each exercise. The partial solution of an exercise is a program that contains all parts that are not considered as the crux of that exercise. The students are asked to make complete programs of these partial solutions. In order to enable the students to concentrate on database topics, we keep the program style simple and consistent over all exercises. We also supply data that can be used to test the constructed databases.

5. Contents of the Practical Work

The practical work is organised in five exercises. Some of these exercises are divided in a number of distinct parts. The main purpose of the first exercise is to get acquainted with functional programming. The next exercise is the construction of an ad hoc database. Due to the embedding of this exercise a relational model-like storage structure will be used by the students. The queries will be similar to relational calculus expressions. The third exercise is the construction of a relational DBMS with queries in relational algebra. In the fourth exercise this data model will be manipulated entirely by a subset of SQL. In the last exercise an existing interpreter for an imperative language is extended by commands to control a hierarchical database.

For each of these exercises we discuss the goal, the question, the given support and the structure of the solution in detail. We made the structure of all programs as consistent as possible. We also used similar applications of the developed DBMSes whenever possible.

Exercise 1: Introduction to Functional Programming

The main purpose of this exercise is to make students sufficiently acquainted with functional programming to make the database programs. We emphasis on IO, the meaning of list comprehensions and working with a program state.
After a large number of examples and simple programs constructed in interaction with the teacher we ask the students to write three small programs.
Part a: Interactive Palindrome Checker
The goal of this part is to make students familiar with simple list manipulations and IO, both as list of characters (`$-`) and as list of values (`$+`). The students should write two programs that check whether lines entered as input are palindromes. One of these programs accepts one list of characters as input. The other takes a list of lines, list of list of characters, as input.
Part b: Pythagorean Triangles.
This part is meant to make students aware of the meaning of ZF-expressions and the advantages of using them. The exercise consists of writing functions that yield the same list of Pythagorean triangles as the given list comprehension.

Students are encouraged to use list comprehensions as much as possible during this course. As an introduction many examples are developed together with the students on the blackboard.
Part c: Reverse Polish Notation Calculator.
The purpose of this part is to teach students to work with a program state, algebraic data types and formatted input ($+ in Miranda). In order to do this, the students have to write an interpreter for a list of statements in reverse polish notation. We help the students by giving them appropriate data types and a description and the type of the functions to implement.

Exercise 2: Ad hoc Database

The topic of this exercise is the construction of a small ad hoc database to store information about books and their authors. The students have to define the state and a number of manipulation functions. We omit details of the attributes to store and manipulations to implement.

The database can be constructed along the same lines as a telephone database as specified in [Diller 94] shown as example. This guides students towards a tailor made relational model. We supply the types of the functions to implement, the command “loop” and data to fill and test the constructed database.

```
command == db -> (output, db)

bibl :: output
bibl = fst (interpret $+ emptydb)

interpret :: [command] -> command
interpret (c: cs) db
```

```
 = (out ++ outs, db2)
   where (out, db1)  = c db
         (outs, db2) = interpret cs db1
```

Using structuring primitives like Monads [Wadler 92, Jones 93] it is possible to define the function `interpret` a little more compact. We use the definition as shown to keep the function as simple as possible for the students.

The state can be defined as:

```
db     == ([author], [wrote], [book])
author == (ssn, name)
wrote  == (ssn, isbn)
book   == (isbn, title, sold)
ssn    == num
isbn   == num
sold   == num
name   == [char]
title  == [char]
```

We show two examples of database manipulations. First the function to add an author. This function checks the consistence of the SSN number as key. The second example is the query to find the authors of the book(s) with the given title.

```
addAuthor :: ssn -> name -> command
addAuthor ssn name db
 = ("Error: author exists", db ), if member ssns ssn
 = (""                    , db'), otherwise
   where (as, ws, bs) = db
         db'  = ((ssn, name): as, ws, bs)
         ssns = [ssn | (ssn, name) <- as]

findAuthors :: title -> command
findAuthors title db
 = (showAuthors as', db)
   where (as, ws, bs) = db
         as' = [(ssn, name)| (bisbn, btitle, bsold) <- bs;
                              btitle = title;
                              (wssn, wisbn) <- ws;
                              wisbn = bisbn;
                              (ssn, name) <- as;
                              ssn = wssn]
```

Note that the list comprehensions have many similarities with expressions in relational calculus. An important difference between relational calculus and ZF-expressions in Miranda is that calculus just gives a tuple definition and a predicate, while the list comprehensions are an algorithm to compute the tuples. All other manipulation functions have the same structure.

Exercise 3: Relational Algebra

In this exercise the students construct their first relational database management system. Instead of a fixed set of relations of known types, as in exercise 2, an arbitrary number of relations is used containing attributes not determined at compile-time. This requires an other approach to define the state. We supply the following definitions.

```
database  == [(tablename, table)]
tablename == [char]

table     == (schema,[tuple])
schema    == [attributename]
attributename == [char]

tuple     == [attribute]
attribute ::= String [char] | Num num | Bool bool | Null
```

Queries to this DBMS are written in relational algebra. The basic operators in algebra are the union, difference, cross product, selection of tuples and projection of attributes. In addition we define operators for the natural join, renaming of attributes, unique (to remove duplicates from the multi-set) and some aggregate functions. Queries are represented by an algebraic data type and contain the abstract syntax tree.

```
query     ::= Table      tablename            |
              Union      query     query      |
              Difference query     query      |
              Cross      query     query      |
              Project    schema    query      |
              Select     condition query      |
              Join       query     query      |
              Rename     [attributename] query|
              Unique     query                |
              Aggregate  schema [(attributename, aggregate)] query

aggregate ::= Sum     attributename           |
              Product attributename           |
              Average attributename           |
              Min     attributename           |
              Max     attributename           |
              Count
```

Conditions come in a number of obvious forms.

```
condition ::= NOT condition                   |
              AND condition  condition        |
              OR  condition  condition        |
              LT  expression expression       |
              …
```

For example, the query $\pi_{a,b}\ \sigma_{a=42}$ (R * T) is represented as the following data structure.

```
Project ["a", "b"] (Select (EQ (Attr "a") (Const (Num 42)))
                           (Join (Table "R") (Table "T")))
```

A `query` is interpreted by the function `retrieve`. This function recursively descends the data structure and calls the appropriate function.

```
retrieve :: query -> db -> table
retrieve query db =
 ret query
 where ret (Union q1 q2) = union (ret q1)(ret q2)
       …
       ret (Unique q)         = unique (ret q)
       ret (Table t)          = lookup emptytable db t
       ret (Aggregate as f q) = groupby as f (ret q)
```

Students should implement the functions which define the semantics of the relational algebra operators:

```
cross      :: table  -> table -> table
difference :: table  -> table -> table
join       :: table  -> table -> table
project    :: schema -> table -> table
rename     :: schema -> table -> table
union      :: table  -> table -> table
unique     ::           table -> table
select     :: (schema- > tuple -> bool) -> table -> table
```

The implementation of these operators using list comprehensions is straightforward and very similar to the definition of the semantics of the operators in set theory. We show some examples (remember that each table consists of a Miranda tuple containing the list of attribute names and a list of database tuples):

```
cross (atts1, tuples1) (atts2, tuples2)
 = (atts1 ++ atts2, [t1 ++ t2 | t1 <- tuples1; t2 <- tuples2])

difference (atts1, tuples1) (atts2, tuples2)
 = (atts1, tuples1 -- tuples2)

select pred (atts, tuples)
 = (atts, [ t | t <- tuples; pred atts t])

union (atts1, tuples1) (atts2, tuples2)
 = (atts1, tuples1 ++ tuples2)
```

To practice writing queries in relational algebra the students should formulate a number of expressions. Since the students are familiar with the data structure to represent queries, a parser for relational algebra is omitted. The queries are written as Miranda data structure. This has as advantage that the database implementation remains simpler and the Miranda mechanisms for abstraction can be used. As example we show the query that yields a table containing the SSN-numbers and names of authors that wrote a book titled `t`. This title is supplied as argument to the command.

```
findAuthor :: title -> query
findAuthor t
 = (Proj ["ssn", "name"] (Sel c universal))
   where c = EQ (Attr "title") (Const (String t))
         universal = Join (Table "author")
                          (Join (Table "wrote") (Table "book"))
```

Exercise 4: Mini-SQL

The purpose of this exercise is to teach the semantics of SQL statements and to practise in writing these statements. To achieve this goal the algebra interface of the previous exercise is replaced by an SQL-interface. We supply the data types involved and the manipulations of the relations. The students should write a query interpreter and a number of SQL-statements.

Since SQL is an enormous large language (only the syntax definition of the core part of SQL2 in BNF takes 47 pages [Melton 93]), it is clear that we must impose severe restrictions here. It is possible to define a small sub-set of SQL, called mini-SQL, that introduces a large part of the features of SQL. We decided to include many possibilities to express queries and to omit the automatic constraint checking and many fancy attribute types.

The SQL based DBMS is constructed on the very same basis as the relational DBMS of the previous exercise. The entire DBMS state remains unchanged. The manipulation of this state will now be done by the following mini-SQL commands.

```
print       :: query -> command
createtable :: tablename -> schema -> command
droptable   :: tablename -> command
insertinto  :: tablename -> query -> command
inserttuple :: tablename -> tuple -> command
deletefrom  :: alias -> condition -> command
updatetable :: alias -> [(attributename, expression)]
                     -> condition -> command
```

The data structure to represent the syntax tree of SQL-queries is defined as:

```
query ::=
  Union  query query                                |
  Except query query                                |
  Select distinct [field] [alias] condition         |
  SelectGrouped distinct [(field, yield)] [alias] condition [field]

yield     ::= Copy | Collect bool aggregate
```

```
aggregate ::= Sum | Product | Average | Min | Max | Count

distinct  == bool
alias     == (tablename, tablename)
field     == (tablename, attributename)

condition ::= Not     condition                        |
              And     condition condition              |
              Or      condition condition              |
              Exists  query                            |
              In      [expression] query               |
              Some    expression relation query        |
              All     expression relation query        |
              Compare expression relation expression

relation   ::= Lt | Le | Eq | Ge | Gt | Ne

expression ::= Plus   expression expression            |
               Times  expression expression            |
               Minus  expression expression            |
               Attr   field                            |
               Const  attribute
```

As example we show the command to remove all authors form the relation `author` that has not written any book. In SQL this can be written as:

```
DELETE FROM author a
   WHERE NOT (EXISTS (SELECT *
                      FROM wrote w
                      WHERE a.ssn = w.ssn))
```

This is represented by the following `command`:

```
deletefrom ("author","a")
   (Not (Exists (Select False []
                 [("wrote","w")]
                 (Compare (Attr ("a","ssn")) Eq (Attr ("w","ssn"))
```

We supply the implementation of the SQL-commands apart from the evaluation of queries. Students should write an interpretation function for the data type `query`. This can be done by a recursive descent of the data type `query` similar to the function retrieve in the previous exercise. We suggest to use a rather naive implementation of the queries. First all tables involved are combined to one table by making the cross product. From this table the tuples obeying the condition in the `WHERE` part are selected. Finally, the attributes listed after the keyword `SELECT` must be projected out.

The existence of sub-queries makes this exercise interesting. Students must be aware of the scope of variables and the semantics of the sub-queries in order to make a correct implementation.

Exercise 5: A Hierarchical DBMS

The topic of the final exercise is the hierarchical data model. This is a somewhat old-fashioned data model that is always manipulated with commands that are embedded in an imperative programming language. The data is logically structured in a strictly hierarchical tree. Physically the data is stored in the list by traversing this tree in pre-order. Via the database commands the user is aware of this physical organisation of the data. The hierarchical DBMS and the imperative program communicate by a number of shared variables. The imperative program controls the actions of the DBMS by executing the appropriate commands.

The purpose of this exercise is to teach the students how the data is organised and how it can be manipulated by embedded commands. This goal is achieved by a similar approach as the previous two exercises: we supply a storage structure for the data and an interpreter for a simple imperative language. The students have to extend this imperative language by retrieval commands for the database management system. This means that they define how the pointers in this linear list of records must be moved.

There are two commands that are used to retrieve information from the database: `GET` and `GETPATH`. Only the `GETPATH` command has to be implemented since the `GET` command can be treated as a special case of the `GETPATH` command. The nasty details of this exercise are omitted since they are not relevant for this paper. We show only the structure of program which is similar to the previous exercises.

```
state      == (database, memory)
command    == state -> (output, state)

memory     == [(identifier, value)]
identifier == [char]

database   == (schema, [record])
schema    ::= Recordtype recordtype [identifier] [schema]
recordtype == [char]
record     == (recordtype, [value])

cond :: expression -> [command] -> [command] -> command
cond e then else (database, memory)
 = exec then (database, memory), if result = Bool True
 = exec else (database, memory), if result = Bool False
 = (nobool, (database, memory)), otherwise
   where result = evaluate memory e
         nobool = "cond: " ++ show e ++ "not a boolean.\n"

while :: expression -> [command] -> command
while expr body (db, mem)
 = ("", (db, mem))                         , if result = Bool False
 = exec (body++[while expr body]) (db, mem), if result = Bool True
 = (nobool,(db, mem))                      , otherwise
   where result = evaluate mem expr
```

```
            nobool = "while: " ++ show result ++ "is not a boolean.\n"
```

As a next step the students write some imperative programs to obtain the some information from a database given as example. These programs can be interpreted by the Miranda program of the first part of this exercise.

6. Reactions of the Students

At the start of the course the students are rather sceptical about functional programming. A typical quote: “In Pascal I write an equivalent program in 10% of the time”. During the course this attitude to functional programming changes completely. At the end of the course only a small minority (less than 10%) of the students indicates that they still prefer an imperative programming language for the kind of exercises in this course.

At the end of the course there is a wide spectrum of opinions on functional programming among the students. Some quotes are used to illustrate this:

- “Functional programming is too difficult for me. Too much is happening in one line.”
- “After you have written some useful function it turns out to be in the standard environment.”
- “Contrary to Pascal I have to think before I start programming.”
- “Functional programming gives the practical part of this course additional value.”
- “Why haven't you told this a long time ago?”

There opinion about Miranda is a bit ambivalent. On one hand they agree that it is a simple language with a nice and rather intuitive syntax. On the other hand they complain about the error messages (especially concerning type errors) and that it is slow. Another drawback is that they cannot run it on their PC at home.

7. Discussion

This paper shows how functional programming is used in an introductory database course. After writing a preliminary program in a functional language the students make four exercises related to databases: an ad-hoc database, an interpreter for relational algebra, an implementation of a sub-set of SQL and a hierarchical database. Some queries have to be written using each of these query languages.

The described approach is successful. The results on the topics of the programming exercises are much better in the exams. Especially the students' ability to write queries and the understanding of query evaluation is considerably improved. This is clearly visible in the parts of the written examination that test the students' ability to write queries in various languages. It is also clear that the students have a better understanding of the internal organisation of a DBMS.

Even for students who had no experience with functional programming it was possible to gain useful training in the database field by writing functional programs. As a matter of fact most of the students prefer the functional language in favour of the imperative languages they know.

A number of students have recorded the time spent on the exercises on a day by day basis. At an average they spent 40 hours to complete all implementation exercises. Considering that these are all newcomers in functional programming this shows that the students were able to do some useful work for a database course in a limited amount of time. It is important to mention that the students do not become full fledged functional programmers by making these exercises. We expect that the students will learn more about the database topics when they had some previous training in functional programming.

The given set of exercises has many possibilities for extensions. For example we can add query optimisation, automatic constraint control, efficient table access or a larger part of the SQL-language. Experienced functional programmers can implement a larger part of the programs themselves.

The given data types to represent queries serve as a clear definition of the border between the database system and the applications made with this system. The introduction of syntax and associated parser for the implemented query languages appears to be unnecessary and unwanted.

References

Diller, A.: Z, An Introduction to Formal Methods, Second edition, Wiley, ISBN 0-471-92973-0, 1994.

Elmasri, Navathe: Fundamentals of database systems, second edition, Benjamin Cummings, ISBN 0-8053-1748-1, 1994.

Fraenkel, A.A.: Zu den Grundlagen der Cantor-Zermeloschen Mengelehre. Mathematische annalen, **86**, pp 230-237, 1922.

Jones, M.P.: Gofer. 2.30 release notes, 1994.

Jones, M.P.: A system of constructor classes: overloading and implicit higher-order polymorphism. in: Proceedings FPCA 93, 1993.

Melton, J., Simon, A.R.: Understanding the new SQL: a complete guide. Morgan Kaufmann Publishers, ISBN 1-55860-245-3, 1993

Turner, D.A.: Miranda: a non-strict functional language with polymorphic types. LNCS **201**, pp 1-16, 1985.

Wadler, P.: The essence of functional programming. In: Proceedings of the 19th annual symposium on Principles of Programming Languages, pp 1-14, 1992.

Zermelo, E.: Untersuchungnen über die Grundlagen der Mengelehre. International Bibliography, Information and Documentation, **65**, pp 261-281, 1908.

Appendix A: Course Description

In order to ease the comparison of the courses discussed in this proceedings, we supply a course description in the format proposed by the program committee.

Title of the Course

Databases and File organisation.

Aims of the Course

The goal of the course is to make students aware of the reasons of existence for database management systems and to give a firm introduction to relational databases. We cover the range from high level modelling (ER, EER, OO) to file organisation. The design of relational schemas and the query languages algebra, calculus and SQL are treated. Also the hierarchical and network model are discussed briefly.

Intended kind of students.

The course is intended for second year computer science students. Usually there is also a small number of students from mathematics or physics visiting the course.

Prerequisites for the course

Students are expected to have some knowledge of imperative programming, data structures like B-trees and hash-functions. None of these topics are very heavily used. Imperative programming is used to show how embedded query languages look. We indicate that it might be very useful to use some tree or hash-function to find a specific record in a relation.

Text book used

For the database part of the course we use *Fundamentals of database systems* by Elmasri and Navathe [Elmasri 94]. As introduction in functional programming we use *Miranda: a non-strict functional language with polymorphic types* [Turner 85] and the Miranda on-line manual. We supply copies of the overhead sheets used as handout. These sheets contain many examples of the use of functional programming languages in a database context. In the current iteration of the course we supplied additional material about functional programming.

Duration of the course

The duration of this course is 12 weeks. The average student is expected to spend 160 hours in total on this course. In the current version of the curriculum this is increased to 200 hours.

There are two lecture hours and two hours of tutorial per week. The maximal time that can be spent on preparation for the examination is about 40 hours. This leaves at least six hours to be spent on home work. In the new curriculum this is nine hours.

Assessment of the students

Students are assessed by a written examination. In previous versions of this course a satisfactory mark of the programming exercises was a prerequisite. In the current version the lab assignments (programming exercises and ER-design etc.) determine 33% of the total assessment.