SQL-compiler

Master thesis in Computing Science

By:

Ronny Andersson 710716-6717 CTH

Supervisors:

Håkan Mattsson, Ericsson Telecom AB

Bengt Johansson, Department of computing science

Department of computing science

Chalmers University of Technology and Göteborg University

1998

Abstract

This report describes a SQL-compiler. The compiler translates the query language SQL to the Erlang programming language and the query language Mnemosyne. Two options are provided for the programmer using the SQL-compiler. SQL statements can either be embedded in Erlang code and compiled together with the Erlang program or compiled at run time when the query is evaluated.

The report explains how and why the compiler was developed. There are also some examples and a tutorial showing how the compiler is used. The compiler is a prototype. Hence, there are some recommendations on how to make a product of it.

The compiler was developed for the database manager Mnesia. Mnesia's query language, Mnemosyne, is embedded in the Erlang programming language. The development of the compiler was done in an UNIX environment using Erlang and some tools supplied by OTP.

Sammanfattning

Denna rapport behandlar en SQL kompilator. Kompilatorn översätter frågespråket SQL till programmeringsspråket Erlang och frågespråket Mnemosyne. En tänkt användare av kompilatorn har två användningsätt att välja på. SQL kommandon kan ligga inbäddat i Erlang koden och bli kompilerade tillsammans med Erlang programmet eller kompileras under exekvering när frågan evalueras.

I rapporten förklaras hur och varför kompilatorn gjordes. Det finns också en del exempel och en användarhandledning på hur kompilatorn används. Kompilatorn är en prototyp. Därför finns det en del rekommendationer på hur en produkt kan byggas.

Kompilatorn var utvecklad för databashanteraren Mnesia. Mnesias frågespråk, Mnemosyne, är inbäddat i programmeringsspråket Erlang. Arbetet med kompilatorn skedde i UNIX miljö och huvudredskapen var Erlang och utvecklingsverktyg från OTP.

Preface

This master thesis is made for the Department of computing science, Chalmers University of Technology/Göteborg University. The master thesis is about a project done at Ericsson Telecom AB/OTP. The supervisors of the project were Håkan Mattsson (Ericsson Telecom AB/OTP) and Bengt Johansson (Department of computing science, Chalmers University of Technology/Göteborg University). Christina von Dorrien (Department of computing science, Chalmers University of Technology/Göteborg University) examined the thesis.

I would especially like to thank my supervisor Håkan Mattsson for his help and support. I would also like to thank Dan Gudmundsson, Lars Thorsen, and the rest of Ericsson Telecom AB/OTP for making it an interesting and stimulating environment to work in. Furthermore Bengt Johansson has also been a great help with external feedback.

1. Contents

2. Introduction

The following sections of this chapter describe why this master thesis was done and how the problems were specified. This chapter also includes a short summary about the Erlang programming language.

Chapter 3 describes the different methods that were used during each part of the project and why these methods were chosen. The chapter also gives some answers about why each specific solution of a problem were chosen for each part of the project.

2.1 Erlang

Erlang[2] is a functional language designed for programming concurrent, real-time, distributed fault-tolerant systems. Erlang's features and the advantages with using Erlang according to the book *Concurrent programming in Erlang*[1] are as followed:

- ***Declarative syntax.*** Erlang has a declarative syntax and is largely free from side-effects. The syntax is similar to other functional programming languages like ML or Haskell.
- ***Concurrent.*** Erlang has a process-based model of concurrency with asynchronous message passing. The concurrency mechanisms in Erlang are lightweight, i.e. process require little memory and context switching is really fast.The computational effort of creating or deleting processes and message passing are small.
- ***Real-time.*** Erlang is intended for programming soft real-time systems.
- ***Continuous operation.*** Erlang has primitives which allow code to be replaced in a running system and allow old and new versions of code to execute at the same time.
- ***Robust.*** For example there are tree constructs in the language for detecting run-time errors. These can be used to program robust applications.
- ***Memory management.*** Erlang is a symbolic programming language with a real-time garbage collector. Memory is allocated automatically when required, and deallocated when no longer used. Typical programming errors associated with memory management cannot occur. Errors related to assignments are eliminated by the fact that Erlang uses single assignment.
- ***Distribution.*** Erlang has no shared memory. All interaction between processes is by asynchronous message passing. Distributed systems are easy to implement in Erlang. Applications written for a single processor can easily be ported to run on networks of processors.
- ***Integration.*** Erlang can easily call or make use of programs written in other programming languages. These can be interfaced to the system in such a way that they appear to the programmer as if they were written in Erlang.

In order to work with Erlang commands an Erlang shell has to be started. The method of starting an Erlang shell is given in example 1, as well as the technique of adding two integers. A typical Erlang module and how to be able to run an Erlang function are presented in examples 2 and 3 below.

```
unix> erl
Erlang (JAM) emulator version 4.6

 Eshell V4.6  (abort with ^G)
1> 4+38.
42
2>
```

EXAMPLE 1. How to start an Erlang shell and perform the operation 4 plus 38.

```
-module(factorial).
-export([factorial/1]).

factorial(0) ->
   1;
factorial(N) ->
   N*factorial(N-1).
```

EXAMPLE 2. Erlang module with the function for evaluating the factorial of a number.

```
121> c(factorial).
{ok,factorial}
122> factorial:factorial(30).
265252859812191058636308480000000
123>
```

EXAMPLE 3. How to compile the module and run the function in example 1.

2.2 Background

The Mnesia DBMS (Appendix C) is a part of OTP(Open Telecom Platform) and uses it's own API (application programming interface) and query language. Mnesia is therefore only accessible for systems implemented in Erlang. There is a requirement that Mnesia ought to be open to other programming languages.

2.3 Problem and purpose

In order to access Mnesia from another programming language it would be appreciable to make an ODBC-driver for Mnesia. ODBC is an interface that provides applications with a single API when accessing different DBMSs. An absolute condition when developing such a ODBC-driver is that Mnesia understands SQL[3] commands. A SQL-compiler would compile SQL commands to the commands Mnesia understands.

The main task of the master thesis was to investigate how the proprietary interface of Mnesia's maps to a traditional relational DBMS with a SQL-based interface. The main purpose of the project was therefore to examine the possibilities of making an acceptable SQL-compiler. Acceptable in the sense that the SQL-functionality translated must reach a certain conformance level otherwise an ODBC-driver can not be developed. The most important questions were how big is the part of SQL's functionality that can be applied on Mnesia without making a tremendous job effort? and would the final compiler be to complex and/or slow?.

To answer these questions and look for differences between SQL and Mnesia's query language in general, the task of the project became to develop a prototype compiler. The prototype should be able to pre-process SQL queries in modules, but also compile queries during run-time in an Erlang shell. For implementation of the prototype the Erlang programming language and related tools were said to be used.

3. Methods

This part of the report will explain how different problems have been approached during the development of the compiler. Smaller problems and solutions are mentioned in Appendix B. Also some of the tools that were used are explained in this chapter. Some names of Erlang data types, for example Tuples, Lists and Strings, are used in this section and they are explained in Appendix B.3.1.

This is what each part of the chapter describes:

3.1 - The parsertool used during the development of the compiler.
3.2 - How a syntax checker for SQL strings was developed.
3.3 - The compile time version of the compiler.
3.4 - Adding some features, that Mnemosyne do not support.
3.5 - The run time version of the compiler.

Erlang is a functional language so during the analysis, design and implementation an incremental approach was used, because the process of implementing a small desirable function is relatively fast. Also, in a bottom-up kind of approach, these small functions used as building blocks in other functions. Therefore, a lot of functionality produced fast and can be tested easily. The system structure are generated automatically and optimizing the system is done as the last step.

3.1 Yecc

The Erlang programming language has a parser generator called yecc that is very similar to the widely known yacc[5]. From a file called the grammar file, with grammar rules of a language, yecc produces Erlang code for a parser.

Grammar rules are also called non terminals and they have the same purpose as grammar rules in our language, for example the grammar rule for making a sentence. The opposite of non terminals are terminals and they represent a building blocks of a language like nouns and verbs for our language.

The Yecc uses a syntax in the grammar file similar to Erlang and all the grammar rules must conform to LALR-1(see the vocabulary). The produced file is the parser of a language. Yecc syntax allows each grammar rule to have some Erlang code attached. This specific Erlang code for each rule will be executed during parsing (see example 4). All Erlang code after the grammar rules in the grammar file represent the back of the compiler.

The Syntax of a rule in a grammar file look like this:

```
Rule1 -> Rule2 : Erlang code.
```

Rule1 is a non terminal (grammar rule).
Rule2 can either be non terminals or terminals.

For example the simplest rule for a sentence in our language:

```
sentence -> noun verb : Erlang code.
```

sentence is a non terminal (grammar rule).

noun and verb are terminals

EXAMPLE 4. Yecc syntax.

3.2 Syntax Checker

The first task was to build a compiler that checked the syntax of a SQL-string. In other words designing and implementing the front of the compiler. Building the syntax checker also gave a deeper knowledge about SQL for further use in the work with the real compiler. There are several standards of SQL on the market. But the standard for this compiler was chosen to the SQL92[4] standard.

First there had to be a scanner that divided the SQL-string into a list of tokens (see example below). A token is a way of representing a terminal and in Erlang a token is represented by a Tuple. The development of the scanner was made in two steps. First the scanner extracted words, numbers and symbols by scanning after special symbols in the string, like the symbol for space. Second it could determine what type of words and numbers it scanned, for example determine if a word is a keyword in SQL92. The output of the scanner is the input of the next part of the compiler, the parser.

```
1> rScan:scan(1,"select person.namn from person").
[{'SELECT',1,keyword},
 {idbody,1,'Person'},
 {'.',special,1},
 {idbody,1,'Namn'},
 {'FROM',1,keyword},
 {idbody,1,'Person'},
 {'$end',1}]
2>
```

EXAMPLE 5. Using the scanner in an Erlang shell.

All the manuals that were used explained the syntax of SQL92 in BNF-notation. Therefore, the first step was to translate it into the LALR-1 form of Yecc. Furthermore the yecc syntax does not have the predicate logic OR, so some grammar rules had to be divided into several subrules (see example 6). The Erlang code in the grammar file was to convert keywords in SQL to words with upper case letters and echo the converted string if it was syntactically correct. Otherwise if the syntax was not correct, the parser would generate an error message (see example 7).

BNF-rule means that colour can either be red or green:

```
colour ::= [ green | red ]
```

The same rule in Yecc syntax:

```
colour -> green : Erlang code.
colour -> red : Erlang code.
```

EXAMPLE 6. Convert BNF to Yecc syntax.

```
1> syntax_checker:parse(“select person.namn from person”).
“SELECT person.namn FROM person”
2> syntax_checker:parse(“huga”).
{1,syntax_checker,[“syntax error before: “,[“‘huga’”]]}
```

EXAMPLE 7. Using the syntax checker from a shell.

3.3 Parse transform

The process of compiling Erlang code, by the Erlang-compiler, consists of several passes (see figure 1). For instance the first pass is the Erlang scanner and parser. The output from this pass is something called a list of Erlang forms. Structure and syntax (see Appendix B.4) of a list of Erlang forms consists basically only of lists and tuples (see example 4). List of forms are the internal compiler representation of a complete Erlang module that the compiler uses in later passes.

Erlang code for adding the integers 21 and 33 are:

```
21 + 33
```

The list of forms for this operation are:

```
[{op,1,’+’,{integer,1,21},{integer,1,33}}]
```

(the number 1 represents on which line the Erlang code is in the module)

Erlang code for assigning the variable A1 with the output value of the Erlang function now().

```
A1 = now()
```

The list of forms for this operation are:

```
[{match,7,{var,7,’A1’},{call,7,{atom,7,now},[]}}]
```

(the number 7 represents on which line the Erlang code is in the module)

EXAMPLE 8. How Erlang code are represented in lists of forms.

Erlang has a parse transform feature that enables the programmer to customize the Erlang-compiler. After the Erlang-compiler has scanned and parsed a module with Erlang code it produces a list of forms (see figure 1). This internal representation of a complete Erlang module is the input of the parse transform. For example Mnemosyne, Mnesia’s own query language, has it’s own customized parse transform that optimizes queries during compile-time.

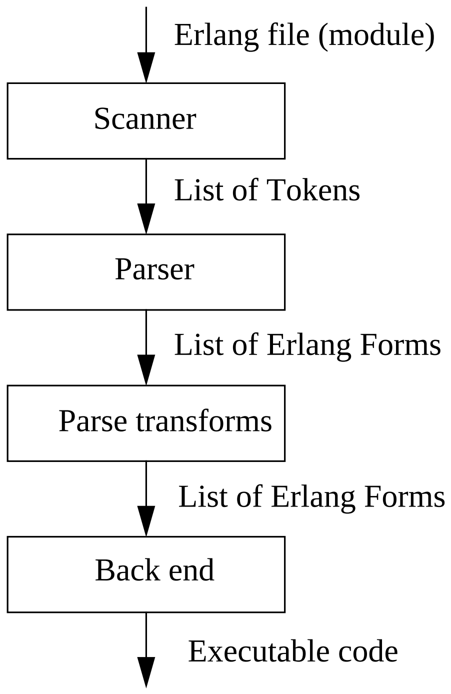

FIGURE 1. The passes done by the compiler.

To integrate the SQL-compiler with the Erlang-compiler in a smooth way, the parse transform feature was chosen. A parse transform that handles SQL commands were implemented. This parse transform searches the list of forms for function calls to a dummy function called sql with a SQL-string as input parameter (see example 7). The function named sql does not exist. Hence, if the parse transform is not used the compiler will generate an error message. The parse transform replaces the function calls in the list of forms with the internal representation of a Mnesia command or Mnemosyne query. The SQL-compiler converts the SQL strings to the internal representation, the Erlang Forms, of the corresponding Mnesia commands or Mnemosyne queries.

```
-module(sqlquery).
-export([make_query/0]).
-compile({parse_transform,intern}).
-record(person, {namn,age,skor}).



make_query() ->
   Handle = sql(“select person.namn
                        from person
                        where person.skor > 40”).
```

EXAMPLE 9. How the dummy function sql are used in an Erlang module.

The first solution to the problem of searching in Erlang forms was to reverse the Erlang grammar and convert each grammar rule with a function. In that way all the possible places where the function call could appear would be detected. This solution was simple and robust, but the implementation was complex and fairly slow. The second and final solution used the fact that Erlang forms only consist of Tuples and Lists (see example 10). Tuples is the Erlang compound data type for storing a fixed number of items and Lists is the Erlang compound data type for storing a variable number of items (see Appendix B.3.1). In that way the search for the function call was even simpler and the implementation was much less complex and faster.

```
[{attribute,1,file,{“./sqlquery.erl”,1}},
    {attribute,1,module,sqlquery},
    {attribute,2,export,[{make_query,0}]},
    {attribute,3,compile,{parse_transform,intern}},
    {attribute,4,
          record,
          {person,[{record_field,4,{atom,4,namn}},
                {record_field,4,{atom,4,age}},
                {record_field,4,{atom,4,skor}}]}},
    {function,6,
          make_query,
          0,
          [{clause,6,
                [],
                [],
                [{match,7,
                      {var,7,’Handle’},
                      {‘query’,
                        7,
                        {lc,7,
                          {cons,
                            7,
                            {record_field,
                              7,
                              {var,7,’Person’},
                              {atom,7,namn}},
                            {nil,7}},
                          [{generate,
                             8,
                             {var,8,’Person’},
                             {call,
                               8,
                               {atom,8,table},
                               [{atom,8,person}]}},
                           {op,10,
                             ‘>’,
                             {record_field,
                               10,
                               {var,10,’Person’},
                               {atom,10,skor}},
                             {integer,10,40}}]}}}]}]},
    {eof,13}]
```

EXAMPLE 10. The Erlang list of forms for the module in example 9.

One approach on designing the SQL-compiler would be to use the compiler directly on the Erlang module and letting the SQL-compiler replace the function calls in the Erlang code with Mnesia commands or Mnemosyne queries. The compiler would then be a kind of pre-processor completely separated from the Erlang-compiler. The disadvantage with this approach, and the reason why it was not used is that the SQL-compiler would have to be able to scan and parse Erlang code in the first compiling pass. It would also lead to that the Erlang-compiler having to scan and parse those commands instead of a simple function calls. Also, there would be some extra costs in opening, modifying and closing the Erlang Module. Using parse transforms, the construction of the SQL-compiler is much simplified, because the list of forms, the internal representation of the Erlang code, has a more organised structure than pure Erlang code.

After the SQL parse transform has modified the list of forms, the list of forms are used as input to the Mnemosyne parse transform. The Mnemosyne parse transform is necessary when Mnemosyne is used. This approach has the advantage that all the optimizing algorithms in the Mnemosyne parse transform are reused, and of course in that way, SQL queries and Mnemosyne queries may be mixed in the same module.

3.4 Adding query features

The current implementation of Mnemosyne has several limitations. For instance inside a Mnemosyne query, expressions are not allowed, only function calls. Therefore the SQL-compiler must convert expressions in SQL queries to function calls in Mnemosyne. The function calls are to a function that have the expression and the variable bindings as input parameters.

Another approach would have been to let the Mnemosyne queries call functions that already exist. But Mnemosyne can only make function calls to a certain level. For example if there was a function, called Add, that added two numbers.Then it is not possible to make those three function calls to Add that is required for adding four numbers (see example below).

```
>Add(1,Add(2,Add(3,4))).
10
>
```

EXAMPLE 11. Using the function Add for evaluating 1+2+3+4 (not possible inside a Mnemosyne query).

3.5 Compiling queries in run time

Mnemosyne has a string interface where a string containing a Mnemosyne query is compiled during run time. This string interface is a simple function call to a function with a Mnemosyne string as an input parameter. The first thing this function does is to convert the string into a list of forms, using a regular Erlang scanner and parser.

When designing a string interface for SQL the same approach as Mnemosyne string interface was used. In that way many of the ideas and solutions in the Mnemosyne string interface could be reused for the SQL string interface.

4. Results

A syntax checker for the complete SQL92 was implemented. All parse conflicts in the given SQL92 grammar were detected and eliminated. The grammar in the grammar file consists of 628 grammar rules and approximately 2500 subrules. This should be compared with for instance the programming language JAVA that consists of only 48 grammar rules. From the grammar file, Yecc generated some 38 000 LOC. This generated parser can not be compiled by the Erlang compiler. The Erlang virtual machine crashes when compiling the parser.

Hence, a subset of the SQL92 grammar was chosen when implementing the prototype. The functionality that were implemented is basically the most commonly used commands in SQL (see Appendix B). About 40% of SQL92s grammar rules were implemented. This is not an estimation of how much SQL-functionality that the prototype manages. It is really hard to estimate how much of SQL's functionality that was implemented.

The functionality tests were successful. All the tests were executed without any crashes and the output values were the desired. The tests were made on a Sun Ultra 167 MHz/128Mb work station and the performance tests gave the following results:

1. The function for converting a SQL string to handle has the execution time of approximately 30 ms for a simple query. This is an overhead of 55% compared with the execution time of Mnemosyne string interface.

2. The execution time for evaluating a SQL query was approximately 5 ms and was exactly the same as the execution time of evaluating Mnemosyne query.

5. Conclusions

The prototype met the requirements of the requirements document.

The most important issue in this project was to determine if it is possible to develop a SQL-compiler for the Mnesia DBMS, and the answer is yes. Furthermore the compiler should not get too slow or too complex. The following conclusions, based on the results, answer why it is possible:

1. Major parts of SQL's functionality are already implemented in the prototype. These parts can be reused when implementing a compiler product. Problems that are not directly related to SQL's functionality are already solved, for instance how to find a SQL string inside an Erlang module. Hence, the developers of the product can concentrate on adding more of SQL's functionality.

2. The complete syntax grammar of SQL92 are implemented in the syntax checker, i.e. the framework of a compiler product are already implemented.

3. The prototype has good performance and the compiler product should get better performance.

4. The prototype can compile a great part of SQL. Even though the amount of time spent on developing the compiler was relatively little (1000 hours).

5. Many parts of SQL's functionality can directly be mapped to Erlang and vice versa For instance the function for determine the length of a string exists in both languages.

The reason why the compiling of the syntax checker crashes is that the Erlang virtual machine can not allocate enough memory. The Erlang virtual machine has not enough address space for compiling the syntax checker.

6. Recommendations

When developing a compiler product the following things should be brought to a discussion:

1. Modify Yecc - The current implementation of Yecc generates two functions with many function clauses. Therefore, the generated code is fairly slow and is really tough for the Erlang compiler to compile. Instead, Yecc should generate many functions with few function clauses.

2. Modify Erlang - If a compiler product is developed, Erlang needs more address space. There is also the possibility of slowing down the rate in which the Erlang virtual machine allocates memory.

3. Modify Mnemosyne - As mentioned have the current implementation of Mnemosyne some limitations compared with SQL. For instance if it was possible to evaluate expressions inside a Mnemosyne query. The implementation of the prototype would be much less complex.

The developers of the product should be able to reuse major parts of the Erlang code in the prototype. The complete SQL92 standard is implemented in the syntax checker. It is therefore a valuable source when developing a compiler product.

7. References

[1] Aho A. V., Sethi R., Ullman J. D.,
Compilers Principles, Techniques, and Tools, Addison Wesley, 1986.

[2] Armstrong J., Virding R., Wikström C. and Williams M.,
Concurrent Programming in ERLANG, Prentice Hall, 1996.

[3] Bowman J. S., Emerson S. L. and Darnovsky M.,
The Practical SQL Handbook, Addison Wesley, 1996.

[4] Date C. J. with Darwen H., A Guide to THE SQL STANDARD,
Addison Wesley, 1997.

[5] Levine J. R., Mason T. and Brown D., lex & yacc, O'Reilly & Associates,
1995.

8. Vocabulary

DBMS - **D**ata**B**ase **M**anagement **S**ystem, software that creates databases and handles data in the databases. Examples of DBMSs are Mnesia and ORACLE.

Handle - Reference to a database query or command.

LALR-1 - Parsing technique and grammar definition when using that technique.
The “LA” is for lookahead, the “L” is for left-to-right scanning of the input, the “R” for constructing a rightmost derivation in reverse, and the “1” for the number of input symbols of lookahead that are used in making parsing decisions[1].

Lists - An Erlang data type.

Mnemosyne - Mnesia’s query language.

ODBC - **O**pen **D**ata**B**ase **C**onnectivity, an interface that provides applications with a single application programming interface(API) when accessing different DBMSs.

OTP - **O**pen **T**elecom **P**latform, a product with the Erlang programming language and some related tools, for instance Mnesia.

Query - *Mnesia meaning*: More complex operations on a database than for example a simple key-value look-up. A query can find all records in a table that fulfils a given property. Queries only deal with extracting data from database tables.
SQL meaning: All operations on a database. Even the operation for creating a database table.

Records - An Erlang data type.

SQL - **S**tructured **Q**uery **L**anguage, a database language.

SQL92 - A version of SQL that is ANSI (ANSI X3.135-1992) and ISO (ISO/IEC 9075:1992) standardised.

Tuples - An Erlang data type.

9. Appendixes

Appendix A. User documentation

A.1 Introduction

This prototype system is a SQL-compiler for converting SQL to Erlang programming language and/or Mnemosyne query language. The purpose for this compiler is to be able to use SQL when working with a Mnesia database. Standard of SQL the compiler are supposed to support is SQL92. The compiler can be used both from an Erlang shell and inside a module (file).

The compiler is not created for the end user of an application. Other programs are supposed to use the functionality of the compiler. Of course the programmer that uses the functionality of the compiler got to have the knowledge of using the compiler.

One important thing to know when using the compiler is that the compiler does not evaluate queries. The compiler only creates handles. In order to evaluate the SQL query the function *eval*, in module sql, must be called. The input to the function *eval* is the handle.

A.2 Using the compiler in a shell

This is how to use the compiler from an Erlang shell:

1. Start an Erlang shell.

2. Start Mnesia with the desired tables.

3. Create a handle to a query by calling the compiler, like this for instance:

```
2> Handle = sql:make_query("select person.namn from person
2> where person.age > 25").
```

4. Evaluate the query by calling the Mnesia transaction function like this:

```
3> mnesia:transaction(
3> fun() ->
3> sql:eval(Handle)
3> end).
{atomic,[["Uwe Krupp"], ["Ron Francis"]]}
```

The second element in the last row above is the result from the query.

A.3 Using the compiler in a Module (file)

How to use the compiler in a module is by making a function call to the function *sql*. The input to *sql* is the SQL string. There also has to be a compile attribute looking like this:

```
-compile({parse_transform,intern}).
```

An example of a module using the compiler can look like this:

```
 -module(sqlquery).
-record(person, {namn, age, skor}).
-export([init/0]).
-compile({parse_transform, intern}).


-record(emp, {namn, idnr}).

init() ->
   A = 1,
   Handle = sql(“select person.namn
                  from person
                  where
                  person.age < 49 - A or person.age = 49”),

 mnesia:transaction(
    fun() ->
         sql:eval(Handle)
    end).
```

Rev A

Appendix B. System documentation

B.1 Requirements document

B.1.1. Background

The specification and background to this system according to Håkan Mattson (Team leader of the Mnesia group, Ericsson Telecom AB/OTP):

Mnesia is a distributed DBMS written in the concurrent functional programming language Erlang. The DBMS has properties that makes it appropriate for demanding applications, such as telecommunications applications, with need of continuous operation and soft real-time properties.

Queries in Mnesia use first order predicate logic (like Prolog), but in a syntax which is suitable for Erlang. The "query list comprehensions" used in Mnesia are taken from the functional languages community. The advantage over embedded SQL, for example, is that the constructs integrate smoothly with the Erlang language.

Open DataBase Connectivity (ODBC) is a kind of standard for access of relational databases. All DBMSs supporting ODBC must have a specific ODBC driver which maps the internals of the DBMS to the standard interface. With a Mnesia specific ODBC driver, Mnesia would appear to be a "standardized" relational DBMS allowing access from existing commercial third party tools and applications written in other programming languages than Erlang.

The main task of the master thesis is to investigate how the proprietary interface of Mnesia's maps to a traditional relational DBMS with a SQL-based interface. In order to obtain an in-depth study, a SQL-compiler should be developed.

B.1.2. Requirements

The requirements from the project management staff at Ericsson Telecom AB/OTP was:

- They wanted a prototype for a SQL-compiler that must be able to run on an Erlang virtual machine.

- The project should take 5 months to complete (10-01-97 => 03-01-98).

The supervisor of the project and also team leader of the team that works with the Mnesia DBMS had the following requirements on the prototype:

- The SQL-compiler should be able to run everywhere Mnesia DBMS are able to run.

- The prototype should have reasonable time performance. Which means that evaluating a small query in real time should not take more than one second. This requirement has it's origin in that Erlang are supposed to have soft real-time performance. In order to make a SQL-compiler product that fulfil the soft real-time demands, the prototype must have reasonable time performance.

- As much as possible of SQL's functionality should be implemented within the time limits of the project.

- The prototype should be implemented in the Erlang programming language.

- During the process of implementing the prototype some of the Erlang tools should be used. Like for instance Yecc, the parser generator.

During the development process of the prototype the following questions should be answered:

- Is it possible to build a SQL-compiler product that supports some level of conformance, that an ODBC-driver for Mnesia DBMS can be implemented?

- If possible, how long time would it take to implement such a compiler?

- How much of SQL's functionality does Mnesia DBMS support directly?

- How hard would it be to implement the functionality in SQL that Mnesia DBMS does not directly support? Would it demand a tremendous job effort? Would the implementation be too slow or too complex?

There were no requirements related to hardware or operating systems, because the Erlang virtual machine supervise all communication with those things.

B.2 System specification (overall)

B.2.1. Compile time system

After the Erlang parser has created the list of Erlang Forms (see chapter 3.3) the Parse transform will produce a new list of Erlang forms. This list will then be the input of the Mnemosyne Parse transform (see figure 2).

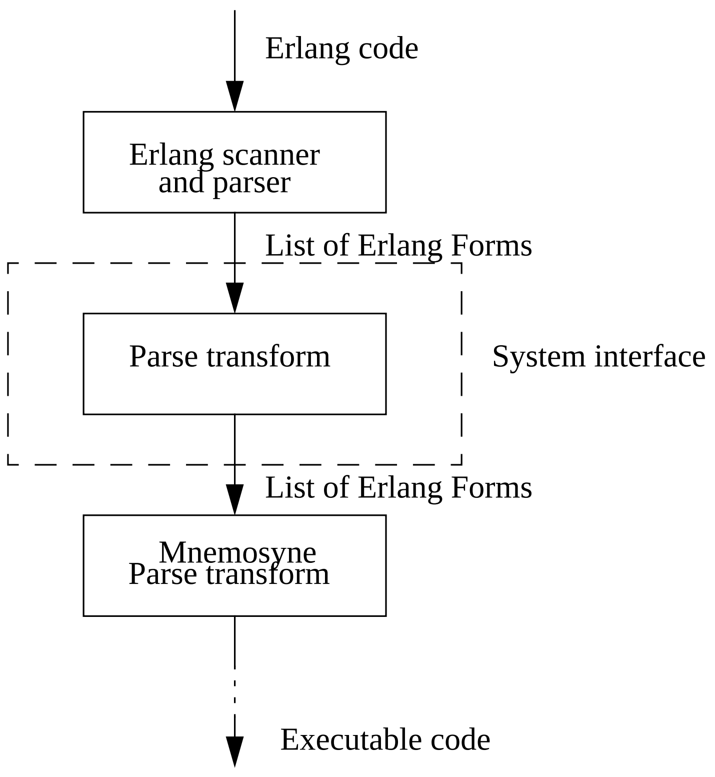

FIGURE 2. Data flow and System interface

The parse transform searches for SQL queries and when a query is found it will be converted to Erlang forms by the SQL-compiler. The SQL-compiler scans and parses the query. The structure of this will be very top-bottom (see figure 3).

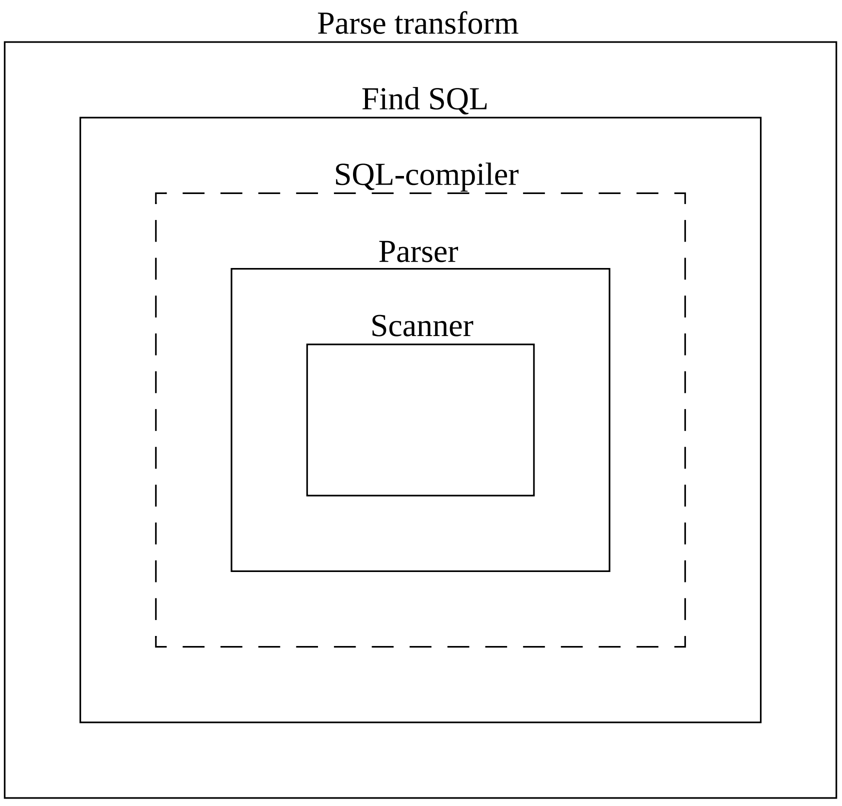

FIGURE 3. Structure of the Parse transform.

B.2.2. Run time system

The SQL-compiler scans and parses the SQL query and produce Erlang Forms. The Erlang Forms are then used by the back end of the Mnemosyne String Interface to create a Handle (See chapter 3.5 and figure 4).

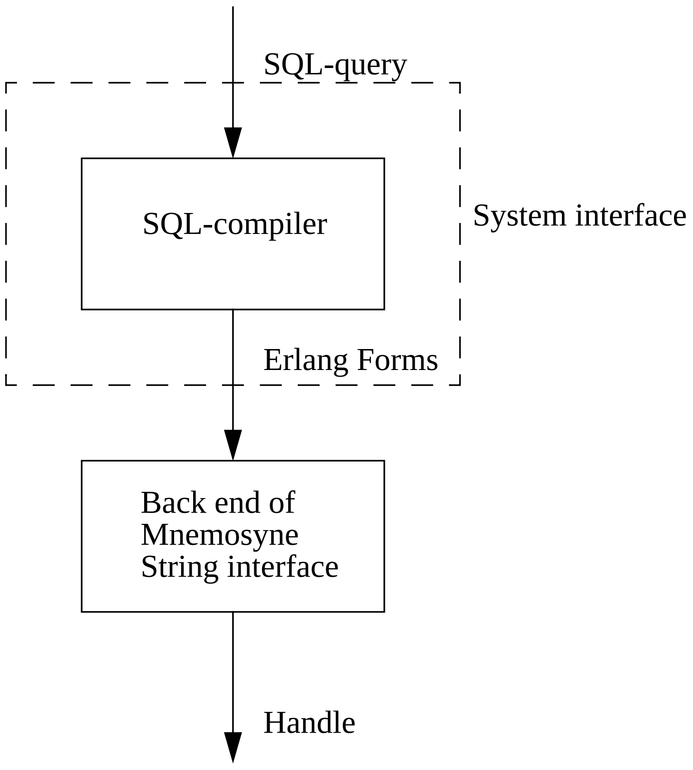

FIGURE 4. Data flow and system interface when running the SQL-compiler i run time.

B.3 System specification (detailed)

B.3.1. Data types in Erlang

Data types according to the book *Concurrent programming in Erlang*[2]:

Constant data types - these are data types which cannot be split into more primitive subtypes

- Numbers - for example: 123, -789, 3.14159, 7.8e12, -1.2e-45.
  Numbers are further subdivided into integers and floats.

- Atoms - for example: abc, ‘An atom with space’, monday, green, hello_world.
  These are simply constants with names.

Compound data types - these are used to group together other data types. There are two compound data types:

- Tuples - for example: {a, 12, b}, {}, {1, 2, 3}, {a, b, c, d, e}.
  Tuples are used for storing a fixed number of items and are written as sequences of items enclosed in curly brackets. Tuples are similar to records or structures in conventional programming languages.

- Lists - for example: [], [a, b, 12], [a, ‘hello friend’].
  Lists are used for storing a variable number of items and are written as sequences of items enclosed in square brackets.

  Components of tuples an lists can themselves be of any Erlang data item - this allows us to create arbitrary complex structures.

The data type Strings is complement to the data type Lists. Strings are basically Lists with characters.

Erlang also has a data structure intended for storing a fixed number of related data items. It is similar to a *struct* in C, or a *record* in Pascal. This data structure is called Records. When referring to a field in a Record, the name of the field is used. This is the big difference compared with Tuples, where position are used when referring to a field. A Record definition can look like this:

```
-record(person, {name, age}).
```

When creating a table in Mnesia there has to be a Record definition like the one above. The table must have the same name as the Record type and the columns in the table must have the same names as the Record field names.

B.3.2. Modules

rScan.erl - Contains the scanner that converts SQL-strings to a list of tokens. The list of tokens are a list of SQL terminals The produced tokens are tuples and can be of following types:

Type	Syntax	Example of String =>token
Bitliteral	{bitliteral, Line, String}	“ B‘10010’ ” => {bitliteral, 3, “10010”}
Hexliteral	{hexliteral, Line, String}	“ H’56EF’ “ => {hexliteral, 3, “56EF”}
Charliteral	{charliteral, Line, String}	“ ‘Hejja’ “ => {charliteral, 2, “Hejja”}}
Natliteral	{natliteral, Line, String}	“ N’Hallå’ “ => {Natliteral, 6, “Hallå”}
DelimitedId	{delimited, Line, String}	“ “_Var” “ => {delimited, 1, “_Var”}
Identifier	{Idbody, Line, Atom}	“Xerxes” => {Idbody, 2, ‘Xerxes’}
Floats	{float, Line, Number}	“3.14” => {float, 2, 3.14000}
integer	{integer, Line, Number}	“42195” => {integer. 5, 42195}
Special	{symbol, special, Line}	“ | “ +> {‘|’, special, 9}
Keyword	{Atom, Line, keyword}	“ select “ => {‘SELECT’, 8, keyword}

The scanner works in two passes. First it divides the string into smaller strings. Then the scanner creates tuples with attributes (see previous table). For instance when determine if a simple string is a Keyword or a Identifier the scanner makes a look-up in an Ets table. Ets tables are a built-in term storage feature in Erlang. The time for looking up a term in Ets tables is constant. So because there are about 320 Keywords in SQL92 this algorithm is efficient compared with searching a list.

select.yrl - The grammarfile for the parser. Only a part of SQL92’s grammar implemented.

select.erl - The Erlang code for the parser generated by Yecc.

syntax_checker.yrl - The grammarfile for the syntax_checker parser. SQL92’s complete grammar implemented.

syntax_checker.erl - The Erlang code for the parser generated by Yecc.

genq.erl - Module with functions that creates Erlang forms.

findsql.erl - Finds the function call sql(“an sql-string”) in a list of forms and converts the SQL string to the corresponding Erlang forms by using the compiler. The algorithm is a deep search of tuples and lists. It searches for a match on the tuple:

{call, Line, {atom, _, sql}, [{_, _, String}]}

String is the SQL-string and is the input to the compiler. The output from the compiler replaces the found tuple in the Erlang forms.

intern.erl - This is the parse transform. It takes the Erlang forms, the output from the Erlang parser, and then call *findsql*. The last step the parse transform perform is to call the Mnemosyne parse transform with the result from *findsql*.

crFunc.erl - Module with functions that creates Erlang forms related to boolean expressions. If there is a an expression in a SQL string the expression has to be converted to a function call. Therefore the module uses the following record types:

-record(pureall,{unpure, pure}).

unpure - Record field with Erlang forms. In case function calls are generated. These Erlang forms are used as input to the function calls.

pure - Record field with Erlang forms. Used when a function calls are not needed.

mEts.erl - Creates an Ets table with SQL92's Keywords.

sqll.erl - Scans and parses a SQL-string and then uses the back end of the Mnemosyne String interface to create a handle.

B.4 Test directions

B.4.1. Functionality

Testing the system's functionality is the most important part of testing according to the requirements. First a database table are created and then will this table be the target of some SQL operations. This testing tries to illustrate a normal use of a data base application. The following groups of SQL statements will be tested:

Create - Create a table in the data base.

Insert - Insert data to the table. In other words add rows to the table.

Select - Extract data from the table. This operation is also known as a query.

Delete - Delete data in the table. In other words remove rows from the table.

Update - Update data in the table. In other words update rows in the table

Drop - Delete a table in the data base.

B.4.2. Performance

In order to measure and test the performance of the run-time version of the compiler some simple SQL queries are tested. The corresponding Mnemosyne queries are also tested. In that way the overhead for the SQL-compiler can be measured. Each query should be executed a 1000 times for increasing the accuracy of the measurements.

B.5 Test protocol

The table that are created and modified will initially look like this:

employee

emp_no	**name**	**salary**	**sex**	**phone**	**room_no**
104440	Andersson Anders	1	m	97760	210
104441	Andersdotter Eva	3	f	97761	211
104442	Persson Per	2	m	97762	212
104443	Persdotter Anna	4	f	97763	213
104444	Jonsson Jon	2	m	97764	214
104445	Jonsdotter Johanna	3	f	97765	215

B.5.1. Functionality

Command:

```
A = sql(“CREATE TABLE employee
                 (emp_no INT,
                  name VARCHAR(20),
                  salary INT,
                  sex CHAR,
                  phone VARCHAR(10),
                  room_no INT)”),
sql:eval(A).
```

Output: {atomic,ok}

OBS! This command only creates an empty employee table in the data base.

Command:

```
A = sql(“insert into employee (emp_no, name, salary, sex, phone, room_no)
        values (104440, ‘Andersson Anders’, 1, ‘m’, 97760, 210)”),
B = sql(“insert into employee (emp_no, name, salary, sex, phone, room_no)
        values (104441, ‘Andersdotter Eva’, 3, ‘f’, 97761, 211)”),
C = sql(“insert into employee (emp_no, name, salary, sex, phone, room_no)
        values (104442, ‘Persson Per’, 2, ‘m’, 97762, 212)”),
D = sql(“insert into employee (emp_no, name, salary, sex, phone, room_no)
        values (104443, ‘Persdotter Anna’, 4, ‘f’, 97763, 213)”),
E = sql(“insert into employee (emp_no, name, salary, sex, phone, room_no)
        values (104444, ‘Jonsson Jon’, 2, ‘m’, 97764, 214)”),
F = sql(“insert into employee (emp_no, name, salary, sex, phone, room_no)
        values (104445, ‘Jonsdotter Johanna’, 3, ‘f’, 97765, 215)”),
mnesia:transaction(fun() ->     sql:eval(A),
                                sql:eval(B),
                                sql:eval(C),
                                sql:eval(D),
                                sql:eval(E),
                                sql:eval(F)
                    end).
```

Output: {atomic,ok}

OBS! After this command the employee table in the data base looks like the table above.

Command:

```
Handle1 = sql(“select employee
                    from employee “),
mnesia:transaction(fun() -> sql:eval(Handle1) end).
```

```
Output: {atomic,[[{employee,104440,”Andersson Anders”,1,”m”,97760,210}],
                 [{employee,104441,”Andersdotter Eva”,3,”f”,97761,211}],
                 [{employee,104442,”Persson Per”,2,”m”,97762,212}],
                 [{employee,104443,”Persdotter Anna”,4,”f”,97763,213}],
                 [{employee,104444,”Jonsson Jon”,2,”m”,97764,214}],
                 [{employee,104445,”Jonsdotter Johanna”,3,”f”,97765,215}]]}
```

OBS! This is select for all the rows in the table and will be used in forthcoming commands to display all the rows in the employee table.

Command:

```
    Handle = sql(“select employee
                    from employee
                    where employee.sex = ‘f’”),
    mnesia:transaction(fun() -> sql:eval(Handle) end).
```

```
Output:  {atomic,[[{employee,104441,”Andersdotter Eva”,3,”f”,97761,211}],
                  [{employee,104443,”Persdotter Anna”,4,”f”,97763,213}],
                  [{employee,104445,”Jonsdotter Johanna”,3,”f”,97765,215}]]}
```

Command:

```
Handle = sql(“select employee.name
                from employee
                where employee.sex = ‘f’ or
                employee.room_no between 213 and 215”),
mnesia:transaction(fun() -> sql:eval(Handle) end).
```

Output:
```
{atomic,[[“Jonsdotter Johanna”],
         [“Jonsson Jon”],
         [“Persdotter Anna”],
         [“Andersdotter Eva”]]}
```

Command:

```
Handle = sql(“delete from employee
              where employee.sex = ‘f’ and
                    SUBSTRING(employee.name FROM 1 FOR 3) = ‘Jon’ “),
Handle1 = sql(“select employee
               from employee “),
mnesia:transaction(fun() -> sql:eval(Handle), sql:eval(Handle1)end).
```

Output:
```
{atomic,[[{employee,104440,”Andersson Anders”,1,”m”,97760,210}],
         [{employee,104441,”Andersdotter Eva”,3,”f”,97761,211}],
         [{employee,104442,”Persson Per”,2,”m”,97762,212}],
         [{employee,104443,”Persdotter Anna”,4,”f”,97763,213}],
         [{employee,104444,”Jonsson Jon”,2,”m”,97764,214}]]}
```

Command:

```
Handle = sql(“update employee set salary = 101
         where employee.sex = ‘f’”),
Handle1 = sql(“select employee
               from employee “),
mnesia:transaction(fun() -> sql:eval(Handle), sql:eval(Handle1)end).
```

Output:
```
{atomic,[[{employee,104440,”Andersson Anders”,1,”m”,97760,210}],
         [{employee,104442,”Persson Per”,2,”m”,97762,212}],
         [{employee,104444,”Jonsson Jon”,2,”m”,97764,214}],
         [{employee,104441,”Andersdotter Eva”,101,”f”,97761,211}],
         [{employee,104443,”Persdotter Anna”,101,”f”,97763,213}]]}
```

Command:

```
A = sql(“drop table employee cascade”),
    sql:eval(A).
```

Output:
```
{atomic,ok}
```

OBS! This command deletes the employee table in the database.

B.5.2. Performance

During the performance tests the employee table (see above) were used. The tests were made on Sun Ultra 167 MHz/128Mb. The command for testing the run-time version of the compiler is:

sql:make_handle(“select employee from employee where employee.salary < 3”).

The corresponding Mnemosyne query is:

```
mnemosyne:string_to_handle(“query [[Employee] || Employee <- table(employe),
                                              Employee.salary < 3] end. “)
```

Test results 1:

Version of the sql - compiler	SQL(ms)	Mnemosyne(ms)
Regular	64.1	17.2
Compiled with pj-flag *	27.9	-

*) Yecc generated parsers can be compiled with an extra compiler option. This optional flag makes the Erlang compiler handle the Yecc generated Erlang code in a special way. The executable code that the Erlang compiler generates with the flag are supposed to be faster.

SQL queries are evaluated with an *eval* function in the *sql* module. Therefore are comparison between *sql:eval* and *mnemosyne:eval* very interesting.

Test results 2:

sql:eval(ms)	Mnemosyne:eval(ms)
5.2	5.2

B.6 Erlang Forms

The table below is an explanation of the data type Erlang forms. Left hand side column of the table represent type names. Right hand side column represent the syntax for the type. Each row for a type in the column represent alternatives of syntax. A type name begins with a upper case letter and other names begins with lower case letters.

TABLE 1. Forms Structure

Name	**Syntax**
Form	Attribute Type_decl Function Rule

Attribute	{attribute, Pos, module, Module}
	{attribute, Pos, export, Farity_list}
	{attribute,Pos, import, {Module, Farity_list}}
	{attribute,Pos, record, {Record, Fields} }
	{attribute,Pos, file, {String, Integer}
	{attribute,Pos, Name, Term}

Module	Atom
Farity_list	[Farity]
Farity	{Atom, Number}
Record	Atom
Fields	[Field]

Field	{record_field, Pos, {atom, Pos, Atom}}
	{record_field, Pos, {atom, Pos, Atom}, Expr}

Term	Atom Integer Float Char String [Term] {Term, Term,......}

Type_decl	{type, Pos, def, Type_header, Utype, Type_constraints}
	{type, Pos, sig, Type_header, Utype, Type_constraints}

Type_header	{Name, Utype_list}
Type_constraints	Constraints []
Constraints	[Constraint]

Constraint	{tcon, Utype, Utype}
	{vcon, Name, Type_tags}

TABLE 1. Forms Structure

Name	**Syntax**
Utype_list	[] Utypes
Utype_tuple	[] Utypes
Utypes	[Utype]

Utype	{utype, Ptypes, []}
	{utype, Ptypes, [Name]}
	{utype, [], [Name]}
	{utype, [], []}

Ptypes	[Ptype]

Ptype	{type, Namn, Utype_list}
	{atom, Name}
	{tuple, Utype_tuple}
	{list, Utype}

Type_tags	[Type_tag]

Type_tag	{tag, Name}
	{atom, Name}
	{tuple, Integer}
	List

Function	{function, Pos, Name, Arity, Clauses}
Clauses	[Clause]
Clause	{clause, Pos, Clause_args, Clause_guard, Clause_body}
Clause_args	Argument_list
Clause_guard	Guard []
Clause_body	Exprs
Exprs	[Expr]
Expr	{catch, Pos, Expr} Expr100

Expr100	{match, Pos, Expr200, Expr100}
	{op, Pos, '!', Expr200, Expr100}
	Expr 200

Expr200	{op, Pos, Comp_op, Expr300, Expr300} Expr300
Expr300	{op, Pos, List_op, Expr400, Expr300} Expr400
Expr400	{op, Pos, Add_op, Expr400, Expr500} Expr500
Expr500	{op, Pos, Mult_op, Expr500, Expr600} Expr600
Expr600	{op, Pos, Prefix_op, Expr_700} Expr_700

TABLE 1. Forms Structure

Name	**Syntax**
Expr700	{call, Pos, Expr_800, Argument_list} Record_expr Expr_800
Expr800	{remote, Pos, Expr_max, Expr_max} Expr_max
Expr_max	Var Atomic List List_comprehension Tuple Expr {block, Pos, Exprs} If_expr Case_expr Receive_expr Fun_expr Query_expr
List	{nil, Pos} {cons, Pos, Expr, Tail}
Tail	{nil, Pos} Expr {cons, Pos, Expr, Tail}
List_comprehension	{lc, Pos, Expr, Lc_exprs}
Lc_exprs	[Lc_expr]
Lc_expr	Expr {generate, Pos, Expr, Expr}
Tuple	{tuple, Pos, Exprs}

Record_expr	{record_index, Pos, Name, Atom}
	{record, Pos, Name, Record_tuple}
	{record_field, Pos, Expr_800, Name, Atom}
	{record, Pos, Expr_800, Name, Record_tuple}
	{record_field, Pos, Expr_800, Name}

Record_tuple	Record_fields
Record_fields	[Record_field] []
Record_field	{record_field, Pos, Atom, Expr}
If_expr	{if, Pos, If_clauses}
If_clauses	[If_clause]
If_clause	{clause, Pos, [], Guard, Clause_body}
Case_expr	{case, Pos, Expr, Cr_clauses}
Cr_clauses	[Cr_clause]
Cr_clause	{clause, Pos, [Expr], Clause_guard, Clause_body}

TABLE 1. Forms Structure

Name	**Syntax**

Receive_expr	{receive, Pos, Cr_clause}
	{receive, Pos, [], Expr, Clause_body}
	{receive, Pos, Cr_clause, Expr, Clause_body}

Fun_expr	{fun, Pos, {function, Name, Number}}
	{fun, Pos, {clauses, Clauses}}

Query_expr	{query, Pos, List_comprehension}
Argument_list	[] Exprs
Guard	Exprs
Atomic	{integer, Pos, Integer} {float, Pos, Float} {atom, Pos, Atom} {string, Pos, String}
Pos	Integer
Arity	Integer
Number	Integer
Name	Atom
Integer	an arbitrary integer number
Float	an arbitrary float number
Atom	an arbitrary atom
Char	an arbitrary character
String	[Char]

Prefix_op	{+, Pos}
	{-, Pos}
	{bnot, Pos}
	{not, Pos}

Mult_op	{*, Pos}
	{/, Pos}
	{div, Pos}
	{rem, Pos}
	{band, Pos}
	{and, Pos}

Add_op	{+, Pos}
	{-, Pos}
	{bor, Pos}
	{bxor, Pos}
	{bsl, Pos}
	{bsr, Pos}
	{or, Pos}
	{xor, Pos}

List	{++, Pos}
	{--, Pos}

TABLE 1. Forms Structure

Name	**Syntax**

Comp_Op	{==, Pos}
	{/=, Pos}
	{=<, Pos}
	{<, Pos}
	{>=, Pos}
	{>, Pos}
	{=:=, Pos}
	{=/=, Pos}

Rule	{rule, Pos, Name, Arity, Rule_clauses}
Rule_clauses	[Rule_clause]
Rule_clause	{clause, Pos, Clause_args, Clause_guard, Rule_body}
Rule_body	Lc_exprs

Appendix C. Mnesia

The following paper is about the Mnesia DBMS.