Uppsala Master's Theses in
Computing Science 293
Examensarbete MN3
2005-08-22
ISSN 1100-1836

Translating SQL expressions to Functional Queries in a Mediator Database System

Markus Jägerskogh

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

A mediator system is a middleware database system that provides uniform queries and views over different wrapped back-end data sources. Amos II is a mediator system using a functional and object oriented query language, AmosQL, to define mediating views of wrapped data sources. This enables users to transparently query the views using AmosQL. The goal of this work is to give the user an SQL interface to Amos II by implementing a pre-processor, SQLFront, from SQL-92 to AmosQL. SQLFront generates AmosQL statements, given SQL statements and an Amos II mapping of a relational schema. Amos II schemas are mapped into SQL by regarding SQL tables as functions in Amos II. SQL queries over tables are translated into AmosQL queries over functions. SQLFront allows mediation through SQL, since SQL tables can be views defined as *derived* AmosQL functions that mediate data from different Amos II sources. In addition Amos II also provides a local store in the mediator system through *stored* functions. SQLFront provides transparent interface to this local store as well, thus providing an SQL interface, including table creation and updates, to the local store.

Supervisor & Examiner: **Tore Risch**

Passed:

Contents

1. Introduction **3**

2. Background **4**

2.1 Database systems 4
2.2 Relational databases 4
2.3 SQL 5
2.4 SQL Verifying test suites 5
2.5 Mediators 5

3. SQLFront Architecture **6**

3.1 Representation of SQL tables 8
3.2 Supported subset of SQL 9

4. Implementation **10**

4.1 Creating tables 10
4.2 INSERT 11
4.3 DELETE 12
4.4 UPDATE 12
4.5 DROP TABLE 13
4.6 COMMIT / ROLLBACK 13
4.7 SCHEMA 13
4.8 SELECT 14
4.8.1 JOIN and NULL-values 15
4.8.2 LIKE / String-handling 16
4.8.3 IN 17
4.8.4 BETWEEN 17
4.8.5 CASE 17
4.8.6 GROUP BY 18
4.8.7 HAVING 20
4.8.8 ORDER BY 21
4.9 Types / Type-conversion 21
4.10 Field name and variable name handling 22

5 Evaluation **22**

6. Mediation **24**

7. Conclusions & Future Work **25**

8. References **26**

A1. Files in the SQL Test suite 27
A2. A larger example 27

1. Introduction

Today there are a lot of different relational databases using SQL as interface language. In addition to this there are an even larger number of other data sources: non-SQL databases, text files and XML, just to mention a few. With an increasing number of different data sources, one would like to be able to make queries over combinations of these data sources, the SQL databases included. At the same time, it would be preferred if this could be done through SQL, since SQL is a wide spread standard for database queries.

It is possible to write an application with the capacity to connect to various data sources and make queries or other kinds of data access to the different kinds of data sources. The application would then have to contain code that combines the data retrieved from the sources and converts the retrieved data to the formats used inside the application. For every new data source new code has to be written. If there are many applications and many kinds of sources the number of interfaces grows fast. Furthermore, the application would contain code that joins accessed data and other services that database management systems (DBMSs) provide. To simplify the applications and minimize the number of interfaces a solution is to introduce an intermediate level of software between databases and their use in applications. Such software is often referred to as *mediators* [10]. A mediator is a module that runs between applications and data sources, offering a single query interface to different data sources and hiding data heterogeneity and query processing from the application.

Amos II [7][8] is a distributed mediator system that uses an object-oriented and functional data model for querying and mediation. Amos II has an object-oriented and functional query language, AmosQL, which is relationally complete.

The main question of this thesis work is: How can we translate SQL into statements in the functional data model of Amos II? Our solution is based on representing SQL tables as Amos II functions. This idea is based on the fact that (stored or derived) functions in Amos II actually correspond to tables (views) in SQL. Since SQL-92 does not support objects we do not need to use the object-oriented parts of AmosQL.

I have implemented a pre-processor to Amos II for executing SQL statements rather than AmosQL. The system, called *SQLFront*, translates SQL statements into internal representations with help from the Amos II kernel in the translation process. The internal representations are then interpreted calling the Amos II executor to produce the desired results. This gives the user a SQL interface to the mediator system. SQLFront allows mediation through SQL, since SQL tables can be views defined with AmosQL that mediate data from different Amos II sources.

To limit the work only simple queries and updates using the SQL-92 standard are supported. SQL-92 does not provide object-orientation and therefore the object-oriented parts of Amos II is not used, only the functional part. In order to access object-oriented data, AmosQL views can be defined in the mediator that remove object abstractions and produce functions without objects.

Amos II can also be used as a stand-alone main-memory DBMS and this work makes it into an SQL-compliant DBMS. Furthermore, storage managers such as BerkelyDB [3] can be wrapped by Amos II [8] in which case this work provides a full disk resident SQL DBMS.

For stand-alone databases SQLFront allows not only data manipulating SQL statements but also data definition through the CREATE TABLE SQL statement.

For evaluation of the system, we use a well-known SQL-92 test suite, the standardized *SQL test suite* [5]. Through this we are able to verify exactly what parts of SQL-92 are supported and that they are working properly. We also use the SQL test suite as a measure of how much of SQL we can handle. We have been able to verify that 72.1% of all statements in the SQL test suite parse. A test of a selected subset shows that about 72 % of the statement executes correctly. The work with SQLFront has been limited not to include: views, nested select statements (including insert into ... select, union, etc), outer joins, NULL values, time and date types, altering of tables, default values. The goal was to be able to parse basic SQL statements with select, project and join. No effort has therefore been put in implementing SQL functions as `TRIM`, `CAST`, `CHAR_LENGTH`, `ABS`, etc.

2. Background

2.1 Database systems

For a long time programs that used data had to store the data themselves in regular files. Databases do in one sense stand as an opposite to "flat files", where applications save their data using some unspecified format. According to [4], there are several conditions that have to be satisfied before a data storage is a database. To begin with the data has to have a known format. It is not enough that the application using the data knows the format; the format must be known to the DBMS. In addition to this, or rather, as a result of this, the format is defined by *metadata* or the *schema*, which is data about data stored in the database. In order to use the term database, the data must be stored, modified and retrieved by the DBMS. The application is no longer allowed to directly fetch or edit the data itself, but has to go through the DBMS. By using a DBMS for data handling, a higher level of security is reached. The risk that some applications make "illegal" operations with the data is eliminated and in case several applications need to do operations with the database at the same time, the DBMS handles the concurrency.

Normally a DBMS requires the user to specify the form of the data before any storage can be done. This includes what items to be stored and what data type these items have. In some systems the physical storage of the items on disk can be specified, as well as if any *indexes* are to be maintained to make it faster to search in the database.

2.2 Relational databases

As we have now seen, there are some criteria for a data storage to be named a database. But there are no rules about how the data in the database is related. In a relational database data is represented as tables. In the relational model the basic unit is the *table*, usually called *relation*. I will use the term *table* to avoid confusion when talking about how tables relate to each other. The table has a set of *attributes* or *columns*. Whenever data is stored in the table, this is done in the form of *tuples* or *rows* by assigning values to all attributes. As an example, an Internet store for bedroom products might create a table named `product`, with the columns `number`, `name` and `price`. In this table we can now store tuples containing information about different products. Every tuple contains data about one product. A simple example is shown in figure 2.2a, where two tuples are

Product		
number	**name**	**price**
001	Pillow	8.45
002	Sheet	2.99

fig 2.2a A simple table

stored; the first one contains information about a pillow and the second one about a sheet. Each row contains information about one product. In the columns on the other hand we find different values for the same attribute; for example all different prices are found in the price column.

2.3 SQL

Structured Query Language (SQL) is a standardized query language for relational databases. SQL is a *non-procedural* or *declarative* language, where the query is expressed in terms of *what* information is wanted, not in *how* to get the information. This makes is easy for the user to access the data, and puts the effort on the database engine (which needs a query optimizer to solve this task effectively). SQL is divided into three sub-parts: The data definition language, the data manipulation language, and the data control language.

The *data definition language* is the part of SQL that is used to manipulate the structures in the database, i.e. the *schema*. This part of SQL is not handling any actual data, but merely how to organize data in tables, using constraints, keys, indices, namespaces etc.

The *data manipulation language* is the part of SQL that is handling data; there are statements for storing, changing, querying, and deleting data in the database. The most used SQL statement, SELECT, belongs to the data manipulation language and is the primitive to search for data fulfilling conditions.

The *data control language* is the part of SQL that is used to control user access to the data. In this part you will find commands to add users and grant or revoke access for them.

SQL was developed by IBM in the 70:s and quickly became a de facto standard querying language for relational databases. Both ISO and ANSI adopted SQL as a standard in the late 80:s.

2.4 SQL Verifying test suites

For testing of SQLFront, we have chosen to use the SQL test suite from National Institute of Standards and Technology [5], which is the official SQL conformance test. The test suite has been released in several versions since 1987, with the current version 6.0, which is a conformance test for SQL92 and hence test that an SQL implementation is conformant to the standard specified in ANSI X3.135-1992 and ISO/IEC 9075:1992. Unfortunately NIST have stopped to develop the SQL test suite, so SQL92 is the latest SQL standard that actually has a conformance test.

The test suite can evaluate both Interactive SQL and Embedded SQL in C and a number of other languages with embedded SQL. We have only used the interactive test, since Amos II currently does not have any support for Embedded SQL.

2.5 Mediators

When building a system where there is a need to combine a large number of autonomous dynamic data sources available on a wide-area network, heterogeneity and data integration is a problem. In cases like this, a good solution is to use *mediators*. Instead of moving the data and transform it to a uniform format[1], a virtual *mediation* layer is added. This way, the data is kept in the sources and it is only necessary to access it on per-need basis. Also, only the

[1] I.e. to a central *data warehouse*.

relevant data is retrieved and combined from the different sources. The mediation layer presents a logically integrated view of the data sources. This layer consists of one or several *mediator* modules, which exploits data for a higher layer of applications and provides a uniform representation of all data sources. Since the data itself is not integrated physically, this view is often referred to as "virtual". The goal of this layer is to simplify, abstract, reduce, merge, and explain data.

In large systems, maintainability is extremely important and because of this the mediator layer is often designed in a modular way, with simple mediators that all are specialized in some way [10]. The higher levels on the other hand, use one or several lower level mediators, to get the needed data. This hierarchy of mediators results in that there is no single global view of the data, instead there might be several mediators to choose from, all sharing some part of the data. Because of this, it is necessary that the mediators present data about themselves. A convenient solution is to let some mediators only handle meta-data about other mediators. An important task for the mediator, besides retrieving data from multiple sources, is to present the mediated data using a common representation, the *Common Data Model*, CDM.
In 1997 the main tasks of a mediation layer was specified [10]:

- accessing and retrieving relevant data from multiple data sources,
- abstraction and transformation of the retrieved data into a common representation and semantics,
- integration and matching of the homogenized data,
- reducing the integrated data by abstraction.

2.6 Amos II

Amos II (Active Mediator Object System) [8] is a distributed mediator system that uses a functional data model and has a relationally complete functional query language, AmosQL. Through its distributed multi-database facilities many autonomous and distributed Amos II peers can interoperate. Functional multi-database queries and views can be defined where external data sources of different kinds are translated through Amos II and reconciled through its functional mediation primitives. Each mediator peer provides a number of transparent functional views of data reconciled from other mediator peers, wrapped data sources, and data stored in Amos II itself. The composition of mediator peers in terms of other peers provides a way to scale the data integration process by composing mediation modules. The Amos II data manager and query processor are extensible so that new application oriented data types and operators can be added to AmosQL, implemented in some external programming language (Java, C, or Lisp). The extensibility allows wrapping data representations specialized for different application areas in mediator peers. The functional data model provides very powerful query and data integration primitives which require advanced query optimization.

3. SQLFront Architecture

Figure 3a illustrates the architecture of SQLFront.

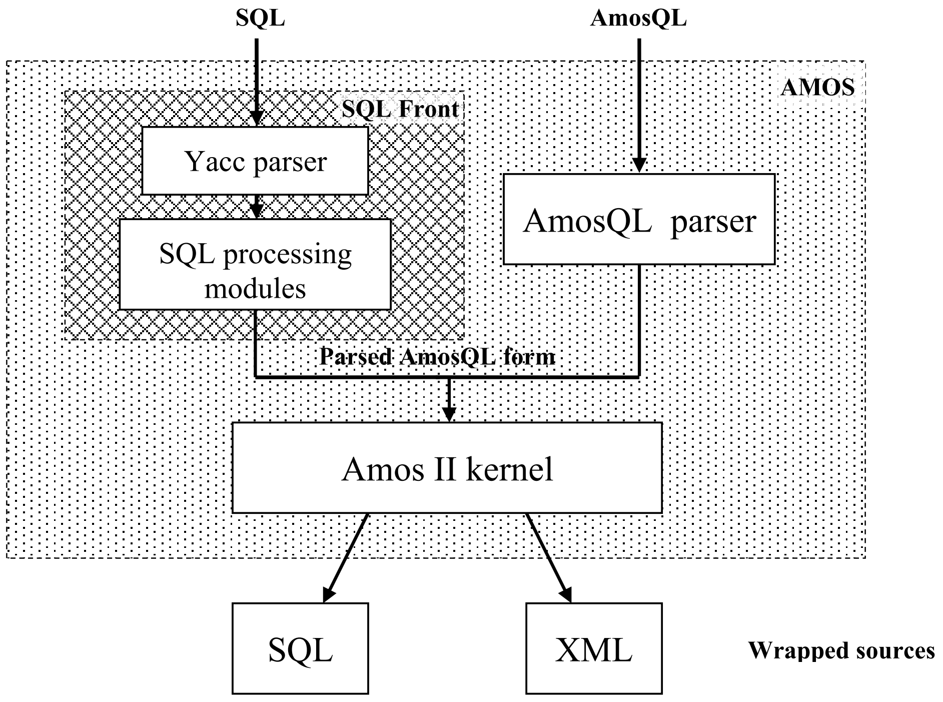

fig 3a - The architecture of SQLFront

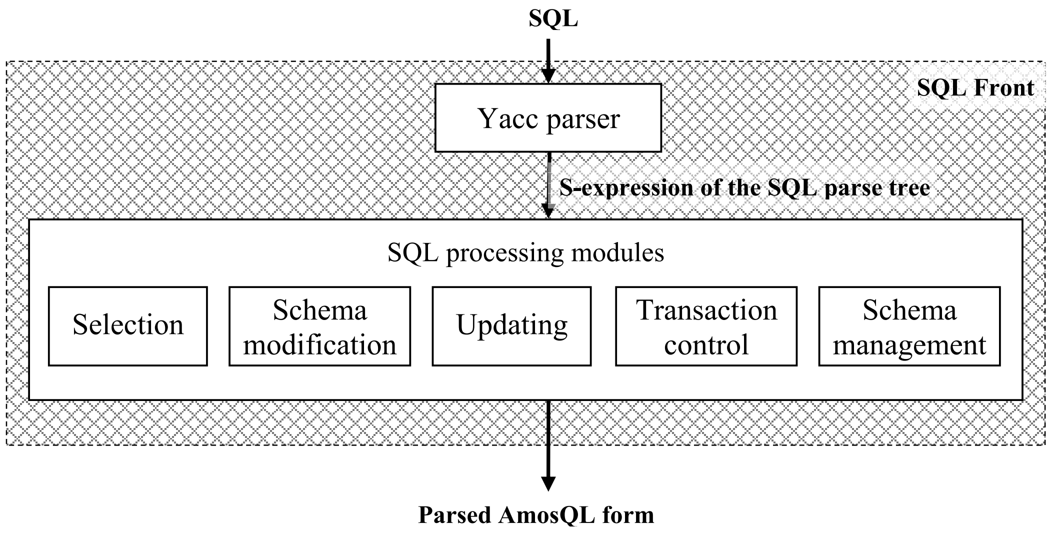

fig 3b - The SQL processing modules in SQLFront

Due to the construction of Amos II, the best way to do the translation is to divide it into two steps. Amos II is based on a C kernel, extended with functionality in Lisp. When an AmosQL query is to be executed, it is first parsed into a parsed query represented as a Lisp S-expression, which is then executed. In this project a similar approach is used; the SQL statement is first parsed into a Lisp macro call that, when executed, constructs a functional query and runs it immediately.

As seen in the figures, SQLFront translates SQL expressions in two steps before execution. In the first step the SQL statement is analyzed by a YACC parser and a parse tree is built. In the second step, the *SQL processing modules*, the parse tree for the SQL statement is transformed into a Lisp S-expression [9] with the same syntax as used by the AmosQL parser to represent the parse tree of an AmosQL statement. When doing this transformation, the meta-data and internal data in Amos II is used to find field names etc. that are needed. The resulting S-expression is immediately sent to the Amos II kernel for execution by its internal Lisp interpreter [6] after the transformation is finished. Lisp macros provide the mechanism for rewriting generated S-expressions into S-expressions executable by the kernel.

As an example, the SQL statement:

```
SELECT NAME, AGE FROM PERSON;
```

is parsed by the YACC parser into an SQL parse tree represented as the S-expression:

```
(SQL-SELECT ((COLUMN nil nil NAME) (COLUMN nil nil AGE)) FROM ((nil
PERSON))
```

The SQL processing module "Selection" (Lisp macro SQL-SELECT) then transforms this into the following S-expression:

```
(OSQL-SELECT (#PERSON.NAME #PERSON.AGE)
 FOREACH ((INTEGER #PERSON.ID) (CHARSTRING #PERSON.NAME)
          (INTEGER #PERSON.AGE))
 WHERE (= (#PERSON #PERSON.ID) (TUPLE #PERSON.NAME #PERSON.AGE)))
```

Another example, the SQL statement:

```
INSERT INTO PERSON (ID, NAME, AGE)
   VALUES (1, 'Markus', 30), (2, 'Elin', 26);
```

is parsed by the YACC parser into the S-expression:

```
(SQL-INSERT PERSON (ID NAME AGE)
    VALUES (1 "Markus" 30) (2 "Elin" 26))
```

The SQL processing module "Updating" (Lisp macro SQL-INSERT) then transforms this into the S-expression:

```
(PROC-BLOCK
   (ADD-FUNCTION #PERSON (1) ("Markus" 30))
   (ADD-FUNCTION #PERSON (2) ("Elin" 26))
)
```

The S-expression is sent to the Amos II kernel for evaluation.

We can also see that SQLFront is actually not translating SQL into AmosQL; it translates directly to an S-expression in Lisp that evaluates the query using the query processing engine of Amos II.

Section 5 describes how different SQL constructs are implemented.

3.1 Representation of SQL tables

The base concept in relational databases is the *table*. Tables are the containers where data is stored and all commands in SQL have something to do with the tables; creating tables, updating tables, creating views over tables, setting user rights for tables, etc. The most used operation is querying tables. Because of this, the representation of tables is a very important issue.

Earlier attempts to write an SQL parser for Amos II [8] has tried to map a table in SQL to a *type* in Amos II, where the columns were mapped to *functions* over the type. This approach has led to a number of problems, especially when it comes to inheritance and other object oriented specific matters.

In the present solution, a table is instead mapped to a function of tuples in Amos II. The functional dependency of the keys in a table is turned into an explicit function mapping between arguments and a corresponding result tuple in Amos II. A function in Amos II is defined with arguments and result as:

```
CREATE FUNCTION fn_name(type arg1, type arg2, ...)->
                <type res1, type res2, ...> as ...;
```

The corresponding SQL table would have the definition

```
fn_name(arg1 type, arg2 type, ..., res1 type, res2 type, ...)
```

where arg_1, arg_2, … are keys.

The concept of a function in Amos II is rather wide; there are *derived functions*, which are similar to views in SQL, *foreign functions*, which are defined in a programming language, *stored procedures*, which are defined using procedural AmosQL statements and thus similar to stored procedures in SQL, and *stored functions* whose content is stored in the database and thus works as a table in SQL.

When using Amos II as a stand-alone database the CREATE TABLE statement of SQL is translated into a stored function definition in Amos II. SQLFront becomes a stand-alone SQL database system.

On the other hand, when using Amos II for mediation, *derived functions* should be used instead and therefore the `CREATE TABLE` statement is not used in this case. A derived function is a function defined by a single AmosQL query. When calling the function, the query is processed to find the requested data from wrapped data sources through Amos II [7][8]. So in SQLFront stored functions represents SQL tables while derived functions define access to Amos II views.

For stored table definitions, when transforming a `CREATE TABLE` statement into a `CREATE FUNCTION` statement, all `PRIMARY KEY`:s are used as arguments to the function and all other columns are put in the result tuple. This is to preserve the functional dependency between the keys and the other columns. The drawback is that this imposes the restriction that keys must precede non-keys in the SQL table definitions.

If the table doesn't have any `PRIMARY KEY`:s, all columns are used as arguments for the function, which then has only a Boolean as result.

Since SQL-92 does not support objects SQLFront translates SQL queries to AmosQL queries only using the functional parts of AmosQL, not involving any objects. The extent of functions mapped to SQL can only contain literals, no objects.

3.2 Supported subset of SQL

SQLFront currently only supports the basic table operations: creation, updating, selection, and deletion. It also supports some transaction control and a simple name-space changing feature. There is no handling of SQL constraints except for primary keys for the tables, and NULL values are not supported. The `SELECT` statement does not support `OUTER JOIN`, but can handle both grouping and sorting. There is no support for nested queries or views. For schemas the <AuthorizationID> and the schema name are used as equivalent, as suggested by [2], even though no checking is done. The only construct for schema handling is a non-SQL standard construct for setting the currently active schema.

4. Implementation

As a first step of the parsing, the SQL statement is parsed into a Lisp macro call. Simplified this step could be explained as building a parse tree as a Lisp S-expression. The formatting of the list is constructed to match the syntax of the statement. As a small example, the SQL statement:

```
SELECT name, age FROM person WHERE age > 7;
```

is parsed into:

```
(SQL-SELECT (NAME AGE) FROM ((NIL _PERSON)) WHERE (< AGE 7))
```

A larger example:

```
SELECT name, COUNT(id_order)
FROM person INNER JOIN order ON person.id = order.id_person
WHERE city='Uppsala'
GROUP BY name
HAVING COUNT(id_order)>3
ORDER BY COUNT(id_order)
```

Is parsed into:

```
(SQL-SELECT (NAME (SQL_COUNT ID_ORDER))
 FROM ((NIL _PERSON)
       (INNERJOIN _ORDER NIL (= PERSON.ID ORDER.ID_PERSON)) )
 WHERE (= CITY "Uppsala")
 GROUPBY (NAME)
 HAVING (> (SQL_COUNT ID_ORDER) 3)
 ORDERBY ((ASC (COUNT ID_ORDER)))
)
```

In this section we will describe how SQLFront translates different SQL statements into S-expressions that execute different processing modules for different kinds of SQL statements.

4.1 Creating tables

The SQL statement for table creation, `CREATE TABLE`, is only applicable when using Amos II as a stand alone DBMS. The statement creates *stored functions.* SQLFront only supports the basic functionality of the `CREATE TABLE` statement. The full statement is accepted, but everything except from the basic functionality is ignored.

The basic form of a `CREATE TABLE` statement in `SQL` is:

```
CREATE TABLE name (field₁ type₁ PRIMARY KEY, field₂ type₂ PRIMARY KEY,
     ..., fieldₖ typeₖ PRIMARY KEY, fieldₖ₊₁ typeₖ₊₁, ..., fieldₙ typeₙ);
```

In AmosQL the corresponding `CREATE FUNCTION` statement looks like this:

```
CREATE FUNCTION name (type₁ field₁, type₂ field₂, ..., typeₖ fieldₖ)->
    <typeₖ₊₁ filedₖ₊₁, typeₖ₊₂ filedₖ₊₂, ..., typeₙ fieldₙ> as stored;
```

As mentioned before, we don't target the translation to AmosQL statements, but rather to the S-expression form that the AmosQL parser delivers. This has several advantages: first, one step of parsing is skipped, second Lisp is excellent in list handling, third the syntax is a little simpler in some cases as for example calls to the `MYIF` function. In order to handle another facility - namespaces or `SCHEMA`:s - all function names are constructed from the table name and the active schema name (or specified schema name) as "schema#table" or, if no schema is selected, only "#table". Let's have a look at an example:

```
CREATE TABLE person(id INT PRIMARY KEY, name CHAR, age INT);
```

Is parsed into the S-expression:

```
(SQL-CREATE-TABLE PERSON (
  (ID (INT) PRIMARY-KEY) (NAME (CHAR)) (AGE (INT))
))
```

SQL-CREATE-TABLE is a Lisp macro that, when executed, rewrites the S-expression into:

```
(CREATEFUNCTION (QUOTE #PERSON)
  (QUOTE ((INTEGER ID)))
  (QUOTE ((CHARSTRING NAME) (INTEGER AGE))) NIL NIL NIL
)
```

representing the final parsed format of the AmosQL statement:

```
create function #person(Integer id)-><Charstring name, Integer age>;
```

One possible problem introduced when translating from tables to functions is that it might be necessary to change the order of the columns when changing the functional dependency of the keys into an explicit function declaration. This is a problem if a table has a `PRIMARY KEY` that comes after a non key column. And if so, the problem will only affect `INSERT`, if column names are omitted, and `SELECT` if asterisk is used.

To avoid this problem, SQLFront has the restriction that all non key columns must come after the PRIMARY KEY columns. This way the reordering is left to the user and nothing is done behind the scene.

Because of the functioning of Amos II, the database will keep track only of key uniqueness when there is only one primary key. If a function is created with only one parameter, Amos II defines the parameter as a key value unless anything else is specified. Because of this, all parameters are defined as being NONKEY if there are no primary key specified. This is the only way key uniqueness constraints are handled.

SQLFront is using the metadata in Amos II and introduces no metadata of its own. This gives the advantage that when creating functions outside of SQLFront, there is no need to create any metadata. The possible drawback of this is that SQL requires some metadata to exist that SQLFront does not have explicitly, but this is solved by creating some foreign functions that translates the metadata in Amos II to the format specified in the SQL standard.

4.2 INSERT

In SQL the `INSERT` statement has several interesting features. In this implementation neither nested queries nor default values are supported. Thus the INSERT statements supported by this implementation are limited to these two variants:

```
INSERT INTO tbl VALUES (val1, ..., valn)
INSERT INTO tbl (field1, ..., fieldn) VALUES (val1, ..., valn)
```

Even though they look very similar, they are different enough to require support in Lisp:

In the first case only the row values are given and should be inserted into the columns in the same order they are given.

In the second case (which is recommended in all SQL books), the order of the fields is not important, as long as the values comes in the same order as the fields they are supported to be put in. When using this syntax, SQLFront matches the field names in the first list with the field names for the function and reorders the values to match the function definition.

The resulting AmosQL syntax for `INSERT` is:

```
add fn(keyfield_1, ..., keyfield_k)=<field_{k+1}, ..., field_n>;
```

SQLFront also checks that all values given matches the type of the fields so an error message will be produced when, e.g.. trying to store a string in an integer field.

4.3 DELETE

The `SQL` statement `DELETE` is used to delete rows in a table. The syntax is:

```
DELETE FROM tbl WHERE <condition>
```

or

```
DELETE FROM tbl
```

In the first case only the rows fulfilling the condition will be deleted, in the latter case all rows will be deleted. This statement was straightforward to implement; there is an analogue statement in AmosQL for functions:

```
remove fn(val_1, ..., val_k)-><val_{k+1}, ..., val_n>;
```

This statement will delete one specific row, where all field values are known. More often only a few field values are know; then SQLFront uses an AmosQL `for each` query like this:

```
for each Type_1 f_1, ..., Type_n f_n
where fn(f_1, ..., f_k)-><f_{k+1}, ..., f_n> and <condition>
remove fn(f_1, ..., f_k)-><f_{k+1}, ..., f_n>;
```

4.4 UPDATE

An update can be performed in different ways. In SQL the statement looks something like this:

```
UPDATE tbl SET field_1=val_1, ..., field_L=val_L WHERE <condition>
```

Quite straight forward, one might think. As usual the condition can be left out with the result that all rows in the table will be updated. In Amos II it wasn't quite that simple. Because the way the functions work separating key values from non-key values, some fields are easy to change and some are not. As described in section 4.1, the fields are divided into two groups, where the fields in the second group, the non-key fields, are functionally dependent on the key fields in the first group. This dependency becomes clear when working with the SET statement:

```
SET fn(keyfield_1, ..., keyfield_k)=<field_{k+1}, ..., field_n>;
```

If this statement is run twice with the same values on all the key fields and we only change some of the values of the non-key fields, the second call will replace the row that the first statement just created. This behaviour is exactly what we expect, but if we would like to change one (or several) of the key-fields, the latter call would instead create a new value tuple and leave the old one untouched in the database. This is on the other hand not what we would expect from an UPDATE. The solution used in this implementation is therefore to split UPDATE into two different cases: In the first case the update changes non-key values only. It is implemented as follows, where some or all of val_L ... val_n have new values passed from the UPDATE statement:

```
for each type $f_1$, ..., type $f_n$
where fn($f_1$, ..., $f_k$)-><$f_{k+1}$, ..., $f_n$> and <condition>
set fn($f_1$, ..., $f_k$)=<$val_L$, ..., $val_n$>;
```

The second case, when key fields are updated, the old updated keys have to be deleted and new ones added. This is done with the code template:

```
for each $Type_1$ $f_1$, ..., $Type_n$ $f_b$
where fn($f_1$, ..., $f_k$)-><$f_{k+1}$, ..., $f_n$> and <condition>
begin
  remove fn($f_1$, ..., $f_k$)=<$f_{k+1}$, ..., $f_n$>;
  add fn($val_1$, ..., $val_k$)=<$val_{k+1}$, ..., $val_n$>;
end;
```

Tables having the entire row as primary key will be translated into a Boolean function:

```
create function fn ($Type_1$ $field_1$, ..., $Type_n$ $field_n$)->Boolean;
```

Such Boolean functions will always use case two, i.e. table keys are always modified.

4.5 DROP TABLE

At first the `DROP TABLE` statement was trivial and was translated directly by the parser to call `delete function` in Amos II for the function, fn, implementing the relational table:

```
DROP TABLE tbl
→
delete function fn;
```

In SQL one might add the optional keywords `RESTRICT` or `CASCADE` at the end of the statement to specify what kind of constraint checking is to be used, but since constraint checking is not used in this implementation, the keywords make no difference. However, some functionality had to be added in SQLFront to take care of schema names. More about schemas in section 4.7.

4.6 COMMIT / ROLLBACK

The two statements for ending a transaction are similar in syntax. `COMMIT` ends a transaction and stores all changes done to the database, whereas `ROLLBACK` ends a transaction and destroys all changes. The SQL syntaxes are:

```
ROLLBACK [ WORK ] [AND [ NO ] CHAIN]
COMMIT [ WORK ] [AND [ NO ] CHAIN]
```

Where the optional keyword `WORK` makes no difference at all and thus is accepted. The key-words "`AND CHAIN`" and "`AND NO CHAIN`" are to control the characteristics for the next transaction, but are not considered as Core SQL and thus not accepted in this implementation. In AMOSQL the corresponding syntaxes are:

```
commit;
rollback;
```

4.7 SCHEMA

The first implementation did not include schema handling at all. But after some testing and examination of the SQL Test suite it became clear that much of the functionality of SQLFront was not tested because many queries in the test suite used the SCHEMA functionality. Because of this we started to investigate the possibility to support the SQL `CREATE SCHEMA` statement. We found a solution that was fairly simple; by adding a special dynamically scoped Lisp variable `*currentschema*` (using Lisp's `DEFVAR` construct) and then use the Lisp macro bellow to implement the statement:

```
(DEFMACRO sql-create-schema (name &rest forms)
  (bquote (LET ((*currentschema* , name))
    ,@ forms)))
```

Since `*currentschema*` is dynamically scoped it will be temporarily rebound when the body forms are evaluated. In addition to this, all Lisp macros implementing SQL statements that are allowed inside a `CREATE SCHEMA` statement must take `*currentschema*` into account.

In this implementation of SQLFront no access rights for different schemas are implemented.

There is also need for a statement to switch between schemas. SQL does not specify how this should be done, but leaves it up to the implementation to solve. According to [2], the naming of schemas and users are by default values for the `CREATE SCHEMA` statement the same if just one of them is specified. In SQL there is no standardized syntax for logging in, but still the schema is dependant on which user is logged in. In our case a very simple construct was added and some minor changes were done to the macros in SQLFront to handle the changes. To set the current schema name we decided to add this construct:

```
SCHEMA name
SCHEMA DEFAULT
```

The latter is used to reset to the default schema (i.e. no schema). The effect of the statement is to set the value of `*currentschema*`. In addition to this, all statements that can handle schema references were updated to take the current schema in account when referencing tables.

The full naming of Amos II functions representing SQL tables in SQLFront is:

```
schemaname#tablename
```

or for the default schema:

```
#tablename
```

4.8 SELECT

The most complex statement in SQL is the SELECT statement and this statements is also the only one used for mediation through SQLFront. Even though it shouldn't be too much trouble to translate nested queries into AmosQL, the limitation not to handle nested queries makes things easier. Many of the syntactic elements in a query are optional, but the full syntax for the SELECT statement without nested queries is:

```
SELECT <column specifications>
FROM <table references>
WHERE <conditions>
GROUP BY <column specifications>
HAVING <grouping conditions>
ORDER BY <column specifications>
```

I will begin by explaining the general translation rules in a simplified manner for the basic query with `SELECT`, `FROM`, and `WHERE` without discussing joining. Joining and the other syntactic elements are discussed in sub-sections below.

The simplest form of query in SQL is to request some column values for all rows in a table:

```
SELECT name, age
FROM person;
```

In AmosQL this would be expressed something like this:

```
select name, age
from Integer id, Charstring name, Integer age
where person(id)=<name, age>;
```

In this translational step, SQLFront first checks that there exists a function corresponding to the table `person`. If the function exists, the parameter names are checked with the column specifications in the SQL query to see that all requested columns exist. If they do, all parameters and their types are fetched to form the `from` clause; then the `where` clause is formed and the query is put together.

Often it is not interesting to get all rows in a table, but rather find a specific row. In SQL this is done by adding a `WHERE` clause to the query, where conditions for row selection are specified. These conditions are just added to the AmosQL `where` clause using the logical and operation:

```
where person(id)=<name, age> and id=7;
```

In SQL there is a short form to request all columns from a table using an asterisk:

```
SELECT * FROM ...
```

or

```
SELECT person.* FROM ...
```

When found by SQLFront, the meta-data in Amos II is used to find the Amos II definition for a requested function. Then this information is traversed to find all fields in their defined order from which the `select` clause is constructed.

Another decision is how column names are to be passed to the macros. In SQL a column can be referenced as `schema.table.column`, `table.column` or simply as `column` in a query. So when translating the query into AmosQL syntax, the macro finds out that the different notations reference the same column and uses one AmosQL variable to represent the column in the generated AmosQL query. In SQLFront the parser transforms a column reference to an S-expression of the form:

```
(COLUMN schema table column)
```

And if not all names are specified, the other names are passed as `nil`:

```
(COLUMN nil table column)
(COLUMN nil nil column)
```

4.8.1 JOIN and NULL-values

In SQL it is possible to query several tables at the same time. This is referred to as *joining*. In a joined query the tables from which information is wanted are listed together with join conditions. If we for instance have a table

```
pet(id int, name char, species_name char,
    ownerID int, PRIMARY KEY(id)),
```

and we would like to list all persons that have a pet and what kind of pet they have, we could write:

```
SELECT person.Name, pet.species_name
FROM person, pet
WHERE person.id = pet.ownerID;
```

In theory a join is based on the Cartesian product of the tables (all rows in `person` are combined with all rows in `pet`) and then the joining conditions are used to throw away the uninteresting results and only keep the interesting ones (where only the persons with the same ID as the `ownerID` stored in the pet table together with the current pet are included in the result). The result after these operations is an *inner join*. The name "inner" is based on the fact that only rows which have a match in the joined table are included. To translate an inner join into Amos II syntax is pretty straight forward. The functionality is the same, only expressed with a different syntax. One major difference in syntax for the chosen representation of SQL tables in Amos II is that in SQL only the columns we are interested in need to be mentioned in the query which makes the queries more compact, whereas our chosen internal representation of tables in Amos II requires all columns to be assigned to a variable. The query above could be expressed like this in AmosQL:

```
select name, species
from Integer id, Charstring name, Integer age,
     Integer petid, Charstring petname, Charstring species,
     Integer ownerID
where pet(petid)=<petname, species, ownerID> and
      person(id)=<name, age> and id=ownerID;
```

SQLFront would produce (and execute) this:

```
(OSQL-SELECT (#PERSON.NAME #PET.SPECIES_NAME)
 FOREACH ((INTEGER #PERSON.ID) (CHARSTRING #PERSON.NAME)
          (INTEGER #PERSON.AGE) (INTEGER #PET.ID)
          (CHARSTRING #PET.NAME) (CHARSTRING #PET.SPECIES_NAME)
          (INTEGER #PET.OWNERID))
 WHERE (AND (= (#PERSON #PERSON.ID) (TUPLE #PERSON.NAME #PERSON.AGE))
            (= (#PET #PET.ID)
               (TUPLE #PET.NAME #PET.SPECIES_NAME #PET.OWNERID))
            (= #PERSON.ID #PET.OWNERID)))
```

As seen in the S-expression above, SQLFront uses periods inside the variable names, something that isn't accepted in AmosQL. This syntax is used to be sure to get unique variable names and to have an easy way to create and match them. Since periods are used in the SQL syntax to separate the table name from the field name, the field name can never contain a period. So by creating variable names as "table.field", we can be sure the names are unique.

4.8.2 LIKE / String-handling

There are some minor differences between how LIKE works in SQL and in AmosQL: First, the wildcards are different, in SQL "_" and "%" are used for wildcards for one character and an unspecified number of characters, in AmosQL "?" and "*" are used. Secondly AmosQL has two functions; LIKE and LIKE_I, which performs case sensitive respectively case insensitive comparison. In SQL different implementations use different solutions for this. The solution used in SQLFront is to always do case insensitive comparison. With a small adjustment in the parser this can be changed to use the AmosQL syntax with LIKE and LIKE_I.

The difference in wildcards gives us a problem. In SQLFront the wildcard translation is done by the parser. The drawback with this solution is that SQLFront has a minor incompatibility with SQL. In SQL it is allowed to pass a column containing the search string containing wildcards:

```
SELECT person.*
FROM person, check
WHERE person.name LIKE check.namefilter;
```

Even if this feature is seldom used in SQL, it is allowed in SQL but not in SQLFront (unless AmosQL wildcards are used in the column containing the search string).

In SQLFront there is no support for the keyword ESCAPE, used to change what escape character to use.

When it comes to string handling there is not so much to say, except listing the two things not supported by SQLFront:

1. String concatenation: "||"
2. Double quotes, "''", or backslash quote, "\'", which both are used to get a single quote in a string, double quotes in the SQL standard, backslash quote in MySQL.

4.8.3 IN

In the current implementation of SQLFront, there is no support for nested queries, so only a simplified support for IN is implemented:

```
SELECT *
FROM person
WHERE age IN (0, 26, 30);
```

The IN statement above would be translated into the OR statement:

```
(OR (= age 0) (= age 26) (= age 30).
```

4.8.4 BETWEEN

The translation of BETWEEN is rather basic. In SQL a query can look like this:

```
SELECT *
FROM person
WHERE age BETWEEN 26 AND 30;
```

And the BETWEEN statement would then be translated into an AND statement:

```
(and (>= age 26) (<= age 30).
```

4.8.5 CASE

The implementation of CASE required more than parsing. In order to make it possible, an AmosQL function had to be created:

```
create function MYIF(Bag p, Bag t, Bag f)->Bag as
  if some(p) then in(t) else in(f);
```

If the first bag, p, is nonempty the elements in the second bag, t, is returned, otherwise the elements in bag f are returned. With this function, the CASE statement could rather easy be

translated, but the CASE statement has two slightly different syntaxes, which could be referred to as CASE and COND:
CASE:

```
SELECT CASE age WHEN 1 THEN 'One' WHEN 2 THEN 'Two' ELSE '> two' END
FROM person;
```

COND:

```
SELECT CASE WHEN (< age 12) THEN 'Child'
            WHEN (< age 18) THEN 'Teenager' ELSE 'Adult' END
FROM person;
```

The CASE statement above is translated as:

```
(MYIF (= age 1) "One" (MYIF (= age 2) "Two" "> two"))
```

And the COND statement above is translated as:

```
(MYIF (< age 12) "Child" (MYIF (< age 18) "Teenager" "Adult"))
```

The MYIF function is overloaded with two more resolvents in order to get the expected behavior. If for example the result is compared with a value, Amos II had problems to understand how to compare a value and a bag.

```
create function MYIF(Bag p, Number t, Number f)-> Number as
  if some(p) then result(t) else result(f);

create function MYIF(Bag p, Charstring t, Charstring f)
    -> Charstring as
  if some(p) then result(t) else result(f);
```

4.8.6 GROUP BY

Grouping is similar to selecting distinct, but gives the possibility to use aggregate functions over the result: counting, summing, maximum, minimum, and average. Even though they are similar, DISTINCT is not suitable for grouping, since this would require every aggregation to be a sub query. Instead a foreign Lisp function is introduced:

```
create function group_by(bag data, integer fcount)->
                     <vector v, vector a> as foreign 'group_by';
```

The function takes two parameters: `data`, which is a selection where the columns to group by precedes the other columns, and `fcount` that specifies how many of the columns that are to be grouped by. The resulting tuple consists of two vectors; the first vector contains the values of the grouped columns, whereas the second vector contains vectors of values of the remaining columns, one vector for each row grouped together. Let's illustrate this with some examples:
Suppose we have four rows in the table `person`:

```
<1, "Markus", 30>
<2, "Fredrik", 33>
<3, "Fredrik", 26>
<4, "Elin", 26>
```

If we then would execute:

```
group_by((select name, age, id from integer id, charstring name,
integer age where #person(id)=<name, age>), 1);
```

The result would be:

```
<{"Markus"},{{30,1}}>
<{"Elin"},{{26,4}}>
<{"Fredrik"},{{33,2},{26,3}}>
```

If we would put age as first column in the select statement:

```
group_by((select age, name, id from integer id, charstring name,
integer age where #person(id)=<name, age>), 1);
```

The result would be:

```
<{26},{{"Fredrik",3},{"Elin",4}}>
<{30},{{"Markus",1}}>
<{33},{{"Fredrik",2}}>
```

If, on the other hand, the second parameter is changed the results would be:
For 0, everything will be grouped in one row:

```
<{},{{30,"Markus",1},{33,"Fredrik",2},{26,"Fredrik",3},{26,"Elin",4}}>
```

For 2, in this case no rows are equal and thus all rows will be grouped by themselves:

```
<{26,"Elin"},{{4}}>
<{30,"Markus"},{{1}}>
<{26,"Fredrik"},{{3}}>
<{33,"Fredrik"},{{2}}>
```

In addition to the `group_by` function, aggregating functions that can work with these tuples are needed. All aggregate functions are implemented as foreign functions in Lisp:
`COUNT` is the simplest function, which just returns the size of the second vector. It does not matter which column you choose to count; all rows will be counted anyhow. The definition of `COUNT` is:

```
create function sql_count(vector a)->integer cnt
  as foreign 'sql-count';
```

The only case when it is important which column is to be counted, is when using `COUNT(DISTINCT column)`. This is implemented as a separate function as:

```
create function sql_countd(vector a, integer field)->integer cnt
  as foreign 'sql-countdistinct';
```

The parameter field is an index of the field to look at in the vector, starting with zero. The function does not care what kind of values appear in the column to be counted, but only counts every value once. Even `nil` are counted.

`MAX` and **`MIN`** are two similar functions with the definitions:

```
create function sql_max(vector a, integer field)->number sum
  as foreign 'sql-max';

create function sql_min(vector a, integer field)->number sum
  as foreign 'sql-min';
```

Both the functions ignore `nil` values, but will return `nil` if there are no non-nil values. `MAX` returns the maximum value and `MIN` the minimum. They use a special comparison function that can compare any type of values. There is no difference on the result if `DISTINCT` is used or not, so `DISTINCT` is allowed, but ignored.

SUM and **AVG** have a very similar function. SUM returns the sum and AVG the average of all rows. None of the functions have a meaning for other input than numbers, so SUM will return zero if no numbers are found in the column (even for nil-values), AVG returns nil if no numbers are found in the column. For both functions DISTINCT will influence the result, so there are a two versions of these functions depending on DISTINCT is used or not, but they have similar definitions. The non-distinct functions are defined as:

```
create function sql_avg(vector a, integer field)->number res
  as foreign 'sql-avg';
create function sql_sum(vector a, integer field)->number sum
  as foreign 'sql-sum';
```

Aggregate functions are only allowed to be used together with the GROUP BY statement in SQL with the exception of queries selecting only aggregate functions:

```
SELECT COUNT(*), AVG(age) FROM person
```

This special type of query is actually a grouping-query where all rows are to be grouped together into one row. SQLFront parses this as:

```
(OSQL-SELECT ((SQL_COUNT A) (SQL_AVG A 1))
 FOREACH ((VECTOR V) (VECTOR A))
 WHERE (AND (= (TUPLE V A)
              (GROUP_BY
                (SELECT (#PERSON.ID #PERSON.AGE)
                 FOREACH ((INTEGER #PERSON.ID)
                          (CHARSTRING #PERSON.NAME)
                          (INTEGER #PERSON.AGE))
                 WHERE (AND (= (#PERSON #PERSON.ID)
                               (TUPLE #PERSON.NAME #PERSON.AGE)))
                ) 0))
            TRUE))
```

4.8.7 HAVING

The HAVING clause very similar to the WHERE clause. The difference between them is that WHERE filters what rows to take into account before the grouping, whereas HAVING filters the rows resulting from the grouping. Because of this HAVING is only allowed together with the GROUP BY clause. Since HAVING is working with the grouped result, aggregate functions may be used in conditions.

Basically, one could say that all search conditions in the WHERE clause are placed in the inner select statement (passed to group_by) and all search conditions in the HAVING clause are placed in the outer select statement.

A small example:

```
SELECT name, AVG(age)
FROM person
WHERE age>25
GROUP BY name
HAVING COUNT(*) > 1;
```

Would be parsed into:

```
(OSQL-SELECT ((VREF V 0) (SQL_AVG A 0))
 FOREACH ((VECTOR V) (VECTOR A))
 WHERE (AND (= (TUPLE V A)
              (GROUP_BY
                (SELECT (#PERSON.NAME #PERSON.AGE)
                 FOREACH ((INTEGER #PERSON.ID)
                          (CHARSTRING #PERSON.NAME)
                          (INTEGER #PERSON.AGE))
                 WHERE (AND (= (#PERSON #PERSON.ID)
                               (TUPLE #PERSON.NAME #PERSON.AGE))
                            (> #PERSON.AGE 25))
                ) 1))
            (> (SQL_COUNT A) 1)))
```

4.8.8 ORDER BY

Even though there are functions in AmosQL for sorting, it turned out that it was much easier to parse if a special sort function was added. This way we could avoid nesting sort functions, which wasn't straight forward to do.

The definition of the function is:

```
create function order_by(bag data, vector order)->object row
  as foreign 'order_by';
```

The first parameter is the result of a query and the second is a vector with the format:

{fieldnumber$_1$, sortorder$_1$, ..., fieldnumber$_n$, sortorder$_n$}

An example of `order_by`:

```
SELECT name, age FROM person ORDER BY name, age DESC;
```

would be parsed into:

```
(OSQL-SELECT (
  (ORDER_BY (SELECT (#PERSON.NAME #PERSON.AGE)
             FOREACH ((INTEGER #PERSON.ID) (CHARSTRING #PERSON.NAME)
                      (INTEGER #PERSON.AGE))
             WHERE (AND (= (#PERSON #PERSON.ID)
                        (TUPLE #PERSON.NAME #PERSON.AGE))))
            (VECTOR 0 "ASC" 1 "DESC"))))
```

4.9 Types / Type-conversion

In Amos II there are rather few built-in data types, whereas SQL has many, so some conversions are necessary. The base rule used here is that all SQL types handling text is converted to CHARSTRING, all types for integers are converted to INTEGER, all types for numbers with decimals are converted to REAL. There are two exceptions from this rule: NUMERIC is converted to NUMBER, since it might be both integers and decimal numbers and BOOLEAN is converted to INTEGER, since SQL always uses the values one and zero. In addition to this the type BOOLEAN in Amos II has special meaning.

In Amos II a function with only one parameter, for example an integer will be defined as:

```
create function f(Integer i)->Boolean as stored;
```

To store a value and remove a value the following statements are used:

```
ADD f(3)=true;
SET f(3)=false;
```

The problem is then if a table like this is created:

```
CREATE TABLE t (i INT PRIMARY KEY, b BOOLEAN);
```

This will be translated into:

```
create function #t(Integer i)->Boolean b as stored;
```

With this definition there is no possibility to store the tuple `<3, false>`, since that operation would be interpreted by Amos II as if we would like to remove the value 3.

There is also a difference between SQL and Amos II in how comparison of numbers is done. Amos II will use integer operations if all numbers in the operation are integers, so `4/3 = 1`, but `4/3.0 = 1.33333`. So if we want the average age of all persons and type:

```
SELECT AVG(age) FROM person;
```

This will result in the integer 28. To get the correct result we must type:

```
SELECT AVG(age+0.0) FROM person;
```

This will give us the expected result 28.75. In other situations SQL has the same behavior, requiring phony decimal numbers to be added for type conversion.

4.10 Field name and variable name handling

In practice the naming of Amos II functions representing SQL tables in SQLFront is:

```
schemaname#tablename
```

And to make naming of variables in queries easy, they are named as:

```
schemaname#tablename.fieldname
```

or

```
tablealias.fieldname
```

This way we have made sure that the names of the variables are unique, since a table cannot have two fields with the same name and a query cannot reference two tables using the same name.

5 Evaluation

The SQL Test suite [5] is a set of files containing SQL expressions. These files are grouped depending on what kind of SQL functionality they are to test. Together with these files there is also a tool for automatic editing of the files. The SQL Test suite can be run automatically using embedded SQL in C (other languages are supported as well), but since currently there is no interface in Amos II for embedded SQL, only the files with SQL expressions are used for testing. Among the files in the SQL Test suite there was a file named `runsql.all`, containing information on which schema each file should be run under.

Several functions has been written to handle the evaluation: a function for reading the SQL files, a Lisp function to perform a test on the statements from one file and some functions for generating statistics. In addition to this, the information found in `runsql.all` was used to construct a list structure with information on all files to test, which schema to use when testing the files etc.

The testing of SQLFront has been done in several steps. The foundation for the tests has been the SQL Test Suite [5]. In the SQL Test Suite there are 368 files with interactive direct SQL statements and a lot of embedded SQL testing. Only the interactive direct SQL statements have been used in the tests of SQLFront, since the embedded SQL does not apply. Appendix 2 shows a list of the file name groups.

In the beginning all these 368 files was run through SQLFront to see how much of the SQL statements that could be parsed without errors. In these tests we were able to verify that about 70 % of all statements in the SQL Test Suite could be parsed. In addition we could see that about 47 % of all statements could be executed without generating any errors. These results does perhaps not say so much of the functionality of SQLFront, since no checking of the result was done, but they were useful for debugging and to verify some functionality.

In the second step a subset of the files was chosen: DML, SDL, XTS, YTS and FLG. The other files were removed since they mainly test things deliberately not implemented in SQLFront. In addition to these some files for schema definition and data loading was used. In total 307 files. These files have been run in the order suggested by the file `sql/runsql.all` to build up a regression test for SQLFront. This far 71 files have been run, checked and added to the regression test manually (i.e. 23.1 % of the subset). The tested files contains 1560 SQL statements (the full SQL Test suite is 4674 SQL statements, so 33.4 % of all SQL statements has been tested and verified manually). Of the tested 1560 statements, 72.9 % or 1137 statements executed correctly.

One thing to note here is that most of the files are designed to test one specific feature of SQL, even though they often use other features "on the way".

When statements not executing correctly were found having other tests depending on the result, these statements were replaced by similar statements giving the same result so the dependant statements wouldn't give wrong result even though they were correct. For instance an `UPDATE` statement with the condition “`WHERE ABS(x)=12`“ would be replaced by the condition “`WHERE x=-12 OR x=12`“ since ABS is not implemented. In addition to this, views were translated into AmosQL manually. For example HU.STAFF2 was translated as:

```
create function HU#STAFF2 (Charstring EMPNUM, Charstring EMPNAME,
                           Real GRADE, Charstring CITY)->Boolean as
  select true
  where HU#STAFF(EMPNUM, EMPNAME, GRADE, CITY);
```

During the test, only 7 statements were found that was expected to execute but didn't. These statements were updates of views where the view had conditions, for instance:

```
CREATE VIEW V_WORKS3 AS
  SELECT * FROM V_WORKS2
  WHERE PNUM = 'P2' OR PNUM = 'P7';
```

This error is because of constraints in current Amos II, preventing updates to views with conditions. There were also 14 statements not executing because the table (or view) they worked with did not exist. The table lists these tables, the reason why they did not exist (the non-implemented feature they were constructed to test) and the number of resulting errors generated due to the missing table.

Table name	**Reason**	**Errors**
CTS1#TT2	INTERVAL	3

CTS1#TABCS	CHARACTER SET	1
CTS1#TTIME3	TIME	1
CTS1#TAB734	NCHAR	1
FLATER#DV1	[didn't have time to write the view]	2
FLATER#VS2 .. #VS6	EXIST	6

The remaining errors are listed in the table below.

Description	**Errors**
`CREATE SCHEMA`	18
`CREATE VIEW`	58
`GRANT`	84
`CREATE TABLE ... CHECK` (supported in most cases)	12
`ALTER TABLE`	2
`CREATE DOMAIN`	7
`CREATE CHARACTER SET`	6
`INTERVAL`	4
`TIME`, `TIMESTAMP`, `DATE` etc	10
`CREATE TABLE ... ON DELETE`	1
`NCHAR`	4
Nested `SELECT`	38
`NULL`	53
`INSERT` statements depending on default values	9
`CREATE INDEX`	2
`SUBSTR()`	1
`ABS()`	1
Extended syntax: SQLFront can't select columns not listed in `GROUP BY` clause	1
Did not prevent access violation (depending on not supporting `GRANT`)	8
`ESCAPE`	1
`EXISTS`, `ALL`, `SOME` and `ANY` (the other occurrences are listed as nested `SELECT`)	4
Trailing spaces in strings (in SQL `'X   '` should equal `'X'`)	1
String quoting (example: the string `"that's"` should be written as `'that''s'`)	1
Comments inside statement	2
USER	74

In addition to this one `CREATE TABLE` statement was changed. In the creation statement for the table `CTS1.DATA_TYPE`, one column was named "DEC", which is a reserved word in SQL-92, SQL-99 and SQL-2003, so this column was renamed to "DES". This has not been counted as an error.

In the SQL Test Suite there are a few statements used to check things outside of the SQL standard. One of the failed statements above (the one marked with italic) is a result of a test of that kind, showing that SQLFront does not support the extension to allow selection of fields not in the group by clause. Thus, this is actually not an error.

6. Mediation

SQLFront itself is not written specifically to do any mediation. The thought with SQLFront is rather to implement an SQL interface to Amos II so the mediation features of Amos II can be reached through SQL. Since there are no ways in SQL to define mediation, AmosQL has to be used for this purpose.

Before any mediation can be done through SQLFront, *derived functions* have to be created in AmosQL, which define the mediators as functional views [8]. A derived function is a function defined by a single AmosQL query. When calling the function, the query is processed to find the requested data through Amos II.

SQLFront makes it possible to make queries on the functions of Amos II if two requirements are fulfilled: first the functions must be defined over literals only, second the functions have to follow the naming conventions of SQLFront.

Hence it is possible to define functions through AmosQL that fulfill the requirements and that mediates data from different data sources, using earlier developed Amos II technique such as [7][8]. The trick is that Amos II allows definition of views that extracts literal data even if Amos II is using objects internally or if the data source is object oriented it self.
I will here show a small example of how functions that extracts literals from objects can be written.

Let's first assume we have the following AmosQL definitions:

```
create Type person;

create function name(person)->Charstring n;

create function age(person)->Integer a;

create function id(person)->Integer i;

create person(id, name, age) instances
         :a(1, "Markus", 30), :b(2, "Fredrik", 33),
         :c(3, "Daniel", 26), :d(4, "Elin", 26);
```

We can then write a derived function to extract literals from these objects:

```
create function obj#persons(Integer id)->
   <Charstring name, Integer age> as
      select n, a
      from Charstring n, Integer a, person p
      where id=id(p) and n=name(p) and a=age(p);
```

This function is now a literal view over the objects, which SQLFront can make queries over:

```
SELECT  age, name
FROM obj.persons
ORDER BY age DESC, name
```

This will return:

```
<33,"Fredrik">
<30,"Markus">
<26,"Daniel">
<26,"Elin">
```

7. Conclusions & Future Work

By this work, it has been shown how to write an SQL parser in SQLFront, for Amos II. This shows that SQL can be expressed as a subset of **AmosQL**, only using literals and Boolean functions. SQLFront was verified by passing a subset of the standard SQL test suite.

The implementation enables SQL access to functional views expressed in AmosQL. Using the mediation facilities of Amos II, these views may retrieve data from different wrapped data sources.

SQLFront includes interfaces to create SQL tables represented as main memory tables in Amos II. By these facilities SQLFront can be used as a main memory SQL database.

Amos II also has facilities for transparent updates of views of mediated data [1]. Using these facilities the update facilities in SQLFront will also be applicable to update mediated source data.

Future work could be to investigate impact of SQL-99 OO concepts on SQLFront.
SQLFront can also be extended to handle sub queries, views, outer joins, and other missing SQL features..

8. References

1. S. Flodin, M. Hansson, V. Josifovski, T. Katchaounov, T. Risch, and M. Sköld: *Amos II user's manual*. (http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html) sec 2.5.1, Uppsala University 2005
2. P. Gulutzan, T. Pelzer: *SQL-99 Complete, Really*. R&D Books, ISBN 0879305681, 1999
3. M.Ladjvardi: *Wrapping a B-Tree Storage Manager in an Object Relational Mediator System* Uppsala Master's Thesis in Computing Science 288, ISSN 1100-1836, 2005. (http://user.it.uu.se/%7Eudbl/Theses/MaryamLadjvardiMSc.pdf)
4. J. Melton, A. Simon: *SQL: 1999 Understanding Relational Language Components*. Morgan Kaufmann Publishers, ISBN 1-55860-456-1, 2002
5. The U.S. National Institute of Standards and Technology (NIST), National Computing Centre Limited (NCC) in the U.K, and Computer Logic R&D in Greece: *The SQL Test Suite, Version 6.0*. (http://www.itl.nist.gov/div897/ctg/sql_form.htm)
6. T.Risch: *ALisp User's Guide*, Uppsala Database Laboratory, 2000. (http://user.it.uu.se/~torer/publ/alisp.pdf)
7. T.Risch and V.Josifovski: Distributed Data Integration by Object-Oriented Mediator Servers. *Concurrency and Computation: Practice and Experience J.* 13(11), John Wiley & Sons, September, 2001. (http://user.it.uu.se/~udbl/publ/ddiooms.pdf)
8. T.Risch, V.Josifovski, and T.Katchaounov: *Functional Data Integration in a Distributed Mediator System*, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003. (http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf)
9. Guy L.Steele Jr.: Common LISP, the language, Digital Press. (http://www.ida.liu.se/imported/cltl/cltl2.html)
10. G. Wiederhold: *Mediators in the Architecture of Future Information Systems*. IEEE Computer, 25(3), 38-49, 1992.

Appendix

A1. Files in the SQL Test suite

The files are named on the form DDDiii.xxx where DDD describes the file content, iii is an integer, assigned serially and xxx gives information on content type. In the tests only files with the three extensions: SQL, SMI and STD are used

Filename	Description	Files
CDR	Tests "integrity enhancement" to SQL	31
DML	Tests data manipulation language	126
SDL	Tests schema definition language via DML	37
ISI	Information schema for Intermediate SQL	7
IST	Information schema for Transitional SQL	7
FLG	SQL Flagger test (syntax extension tests)	6
MPA	Concurrency test, program A	31
MPB	Concurrency test, program B to be run at the same time as program A	14
XOP	X/Open Extension test. For embedded SQL C and COBOL only.	?
XTS,YTS	Intermediate SQL programs donated by the CTS2 SQL2 project	55, 54

A2. A larger example

In this section a series of SQL statements are parsed and executed, showing a possible execution of statements, how they are parsed and the result of the execution.

The statements are shown in three steps: a, b and c, followed by the result (d), as:
a) SQL, b) SQL parsed into S-expression, c) Translated S-expression in AmosQL form, d) Result (when applicable).

```
a) CREATE TABLE person (id INT PRIMARY KEY, name CHAR, age INT);
b) (SQL-CREATE-TABLE PERSON ((ID (INT) PRIMARY-KEY)
                             (NAME (CHAR)) (AGE (INT))))
c) (SET-SQL-IMPLEMENTATION (QUOTE #PERSON) (CREATEFUNCTION
     (QUOTE #PERSON) (QUOTE ((#[OID 9 "INTEGER"] ID)))
     (QUOTE ((#[OID 11 "CHARSTRING"] NAME) (#[OID 9 "INTEGER"] AGE)))
     NIL NIL NIL))

a) CREATE TABLE spieces (id INT PRIMARY KEY, name CHAR);
b) (SQL-CREATE-TABLE SPIECES ((ID (INT) PRIMARY-KEY) (NAME (CHAR))))
c) (SET-SQL-IMPLEMENTATION (QUOTE #SPIECES) (CREATEFUNCTION
     (QUOTE #SPIECES) (QUOTE ((#[OID 9 "INTEGER"] ID)))
     (QUOTE ((#[OID 11 "CHARSTRING"] NAME))) NIL NIL NIL))

a) CREATE TABLE pet (id_pet INT PRIMARY KEY, id_person INT,
                     id_spieces INT, name CHAR(30));
b) (SQL-CREATE-TABLE PET ((ID_PET (INT) PRIMARY-KEY) (ID_PERSON (INT))
                          (ID_SPIECES(INT)) (NAME (CHAR 30))))
c) (SET-SQL-IMPLEMENTATION (QUOTE #PET) (CREATEFUNCTION
     (QUOTE #PET) (QUOTE ((#[OID 9 "INTEGER"] ID_PET)))
     (QUOTE ((#[OID 9 "INTEGER"] ID_PERSON)
     (#[OID 9 "INTEGER"] ID_SPIECES) (#[OID 11 "CHARSTRING"] NAME)))
     NIL NIL NIL))
```

```
a) INSERT INTO person VALUES (1, 'Markus', 30), (2, 'Elin', 26),
      (3, 'Fredrik', 33), (4, 'Daniel', 26);
b) (SQL-INSERT PERSON VALUES (1 "Markus" 30) (2 "Elin" 26)
      (3 "Fredrik" 33) (4 "Daniel" 26))
c) (PROC-BLOCK
     (ADD-FUNCTION #PERSON (1) ("Markus" 30))
     (ADD-FUNCTION #PERSON (2) ("Elin" 26))
     (ADD-FUNCTION #PERSON (3) ("Fredrik" 33))
     (ADD-FUNCTION #PERSON (4) ("Daniel" 26)))

a) INSERT INTO spieces VALUES (1, 'cat'), (2, 'dog'),
      (3, 'budgie'), (4, 'parrot'), (5, 'guinea pig');
b) (SQL-INSERT SPIECES VALUES (1 "cat") (2 "dog") (3 "budgie")
      (4 "parrot") (5 "guinea pig"))
c) (PROC-BLOCK
      (ADD-FUNCTION #SPIECES (1) ("cat"))
      (ADD-FUNCTION #SPIECES (2) ("dog"))
      (ADD-FUNCTION #SPIECES (3) ("budgie"))
      (ADD-FUNCTION #SPIECES (4) ("parrot"))
      (ADD-FUNCTION #SPIECES (5) ("guinea pig")))

a) INSERT INTO pet VALUES (1, 1, 2, 'Spot'), (2, 3, 4, 'Onkel Sam'),
      (3, 2, 2, 'Snuckoms'), (4, 4, 3, 'Carl'), (5, 4, 3, 'Ada'),
      (6, 1, 5, 'Mickey'), (7, 1, 1, 'Masse'), (8, 3, 1, 'Mr');
b) (SQL-INSERT PET VALUES (1 1 2 "Spot") (2 3 4 "Onkel Sam")
      (3 2 2 "Snuckoms") (4 4 3 "Carl") (5 4 3 "Ada")
      (6 1 5 "Mickey") (7 1 1 "Masse") (8 3 1 "Mr"))
c) (PROC-BLOCK
      (ADD-FUNCTION #PET (1) (1 2 "Spot"))
      (ADD-FUNCTION #PET (2) (3 4 "Onkel Sam"))
      (ADD-FUNCTION #PET (3) (2 2 "Snuckoms"))
      (ADD-FUNCTION #PET (4) (4 3 "Carl"))
      (ADD-FUNCTION #PET (5) (4 3 "Ada"))
      (ADD-FUNCTION #PET (6) (1 5 "Mickey"))
      (ADD-FUNCTION #PET (7) (1 1 "Masse"))
      (ADD-FUNCTION #PET (8) (3 1 "Mr")))

a) COMMIT WORK;
b) (COMMIT)
c) (COMMIT)

a) UPDATE pet SET name='Mister' WHERE name='Mr';
b) (SQL-UPDATE PET SET ((NAME . "Mister"))
      (WHERE (= (COLUMN NIL NIL NAME) "Mr")))
c) (OSQL-FOREACH ((INTEGER ID_PET) (INTEGER ID_PERSON)
                  (INTEGER ID_SPIECES) (CHARSTRING NAME))
      (AND (= (#PET ID_PET) (TUPLE ID_PERSON ID_SPIECES NAME))
           (= NAME "Mr")) NIL
     (SET-FUNCTION #PET (ID_PET) (ID_PERSON ID_SPIECES "Mister")))
```

```
a) SELECT person.name, pet.name
   FROM person INNER JOIN
        pet ON person.id = pet.id_person
   WHERE age >= 30;
b) (SQL-SELECT ((COLUMN NIL PERSON NAME) (COLUMN NIL PET NAME))
    FROM ((NIL PERSON) (INNERJOIN PET NIL
       (= (COLUMN NIL PERSON ID) (COLUMN NIL PET ID_PERSON))))
    WHERE (>= (COLUMN NIL NIL AGE) 30))
c) (OSQL-SELECT (#PERSON.NAME #PET.NAME)
    FOREACH ((INTEGER #PERSON.ID) (CHARSTRING #PERSON.NAME)
             (INTEGER #PERSON.AGE) (INTEGER #PET.ID_PET)
             (INTEGER #PET.ID_PERSON) (INTEGER #PET.ID_SPIECES)
             (CHARSTRING #PET.NAME))
    WHERE (AND (= (#PERSON #PERSON.ID)
                  (TUPLE #PERSON.NAME #PERSON.AGE))
               (= (#PET #PET.ID_PET)
                  (TUPLE #PET.ID_PERSON #PET.ID_SPIECES #PET.NAME))
               (AND (>= #PERSON.AGE 30)
                    (= #PERSON.ID #PET.ID_PERSON))))
d) (("Markus" "Spot") ("Markus" "Mickey") ("Markus" "Masse")
    ("Fredrik" "Onkel Sam") ("Fredrik" "Mister"))

a) ROLLBACK WORK;
b) (ROLLBACK)
c) (ROLLBACK)

a) SELECT person.name, spieces.name, COUNT(id_pet)
   FROM person, pet, spieces
   WHERE person.id = pet.id_person AND pet.id_spieces = spieces.id
   GROUP BY person.name, spieces.name;
b) (SQL-SELECT ((COLUMN NIL PERSON NAME) (COLUMN NIL SPIECES NAME)
                (SQL_COUNT (COLUMN NIL NIL ID_PET)))
    FROM ((NIL PERSON) (NIL PET) (NIL SPIECES))
    WHERE (AND (= (COLUMN NIL PERSON ID) (COLUMN NIL PET ID_PERSON))
               (= (COLUMN NIL PET ID_SPIECES) (COLUMN NIL SPIECES ID)))
    GROUPBY ((COLUMN NIL PERSON NAME) (COLUMN NIL SPIECES NAME)))
c) (OSQL-SELECT ((VREF V 0) (VREF V 1) (SQL_COUNT A))
    FOREACH ((VECTOR V) (VECTOR A))
    WHERE (AND (= (TUPLE V A) (GROUP_BY
      (SELECT (#PERSON.NAME #SPIECES.NAME #PET.ID_PET)
       FOREACH ((INTEGER #PERSON.ID) (CHARSTRING #PERSON.NAME)
                (INTEGER #PERSON.AGE) (INTEGER #PET.ID_PET)
                (INTEGER #PET.ID_PERSON) (INTEGER #PET.ID_SPIECES)
                (CHARSTRING #PET.NAME) (INTEGER #SPIECES.ID)
                (CHARSTRING #SPIECES.NAME))
       WHERE (AND (= (#PERSON #PERSON.ID)
                     (TUPLE #PERSON.NAME #PERSON.AGE))
                  (= (#PET #PET.ID_PET)
                     (TUPLE #PET.ID_PERSON #PET.ID_SPIECES #PET.NAME))
                  (= (#SPIECES #SPIECES.ID) #SPIECES.NAME)
                  (AND (= #PERSON.ID #PET.ID_PERSON)
                       (= #PET.ID_SPIECES #SPIECES.ID)))
      ) 2)) TRUE))
d) (("Markus" "guinea pig" 1) ("Markus" "dog" 1) ("Elin" "dog" 1)
    ("Markus" "cat" 1) ("Fredrik" "parrot" 1) ("Fredrik" "cat" 1)
    ("Daniel" "budgie" 2))
```

```
a) SELECT MAX(name), MIN(name)
   FROM pet;
b) (SQL-SELECT ((SQL_MAX (COLUMN NIL NIL NAME))
                (SQL_MIN (COLUMN NIL NIL NAME)))
    FROM ((NIL PET)))
c) (OSQL-SELECT ((SQL_MAX A 0) (SQL_MIN A 0))
    FOREACH ((VECTOR V) (VECTOR A))
    WHERE (AND (= (TUPLE V A) (GROUP_BY
      (SELECT (#PET.NAME)
       FOREACH ((INTEGER#PET.ID_PET) (INTEGER #PET.ID_PERSON)
                (INTEGER #PET.ID_SPIECES) (CHARSTRING #PET.NAME))
       WHERE (AND (= (#PET #PET.ID_PET)
                     (TUPLE #PET.ID_PERSON #PET.ID_SPIECES #PET.NAME)))
      ) 0)) TRUE))
d) (("Spot" "Ada"))

a) SELECT s.name, COUNT(id_pet)
   FROM pet p INNER JOIN
        spieces s ON p.id_spieces = s.id
   GROUP BY s.name
   ORDER BY COUNT(id_pet) DESC, s.name
b) (SQL-SELECT ((COLUMN NIL S NAME)
                (SQL_COUNT (COLUMN NIL NIL ID_PET)))
    FROM ((NIL PET P) (INNERJOIN SPIECES S
      (= (COLUMN NIL P ID_SPIECES) (COLUMN NIL S ID))))
    GROUPBY ((COLUMN NIL S NAME))
    ORDERBY ((DESC (SQL_COUNT (COLUMN NIL NIL ID_PET)))
             (ASC (COLUMN NIL S NAME))))
c) (OSQL-SELECT ((ORDER_BY
      (SELECT ((VREF V 0) (SQL_COUNT A))
       FOREACH ((VECTOR V) (VECTOR A))
       WHERE (AND (= (TUPLE V A)
      (GROUP_BY
        (SELECT (#S.NAME #P.ID_PET)
         FOREACH ((INTEGER #P.ID_PET) (INTEGER #P.ID_PERSON)
                  (INTEGER #P.ID_SPIECES) (CHARSTRING #P.NAME)
                  (INTEGER #S.ID) (CHARSTRING #S.NAME))
         WHERE (AND (= (#PET #P.ID_PET)
                       (TUPLE #P.ID_PERSON #P.ID_SPIECES #P.NAME))
                    (= (#SPIECES #S.ID) #S.NAME)
                    (AND (= #P.ID_SPIECES #S.ID)))
        ) 1)) TRUE))
      (VECTOR 1 "DESC" 0 "ASC"))))
d) ((("budgie" 2)) (("cat" 2)) (("dog" 2))
    (("guinea pig" 1)) (("parrot" 1)))

a) DELETE FROM pet WHERE id_person>2;
b) (SQL-DELETE PET (> (COLUMN NIL NIL ID_PERSON) 2))
c) (OSQL-FOREACH ((INTEGER ID_PET) (INTEGER ID_PERSON)
                  (INTEGER ID_SPIECES) (CHARSTRING NAME))
      (AND (= (#PET ID_PET) (TUPLE ID_PERSON ID_SPIECES NAME))
           (> ID_PERSON 2)) NIL
        (REM-FUNCTION #PET (ID_PET) (ID_PERSON ID_SPIECES NAME)))
```

```
a) SELECT COUNT(*)
   FROM pet;
b) (SQL-SELECT ((SQL_COUNT *)) FROM ((NIL PET)))
c) (OSQL-SELECT ((SQL_COUNT A))
    FOREACH ((VECTOR V) (VECTOR A))
    WHERE (AND (= (TUPLE V A) (GROUP_BY
      (SELECT (#PET.ID_PET)
       FOREACH ((INTEGER #PET.ID_PET) (INTEGER #PET.ID_PERSON)
                (INTEGER #PET.ID_SPIECES) (CHARSTRING #PET.NAME))
       WHERE (AND (= (#PET #PET.ID_PET)
                     (TUPLE #PET.ID_PERSON #PET.ID_SPIECES #PET.NAME)))
      ) 0)) TRUE))
d) ((4))

a) DROP TABLE pet;
b) (SQL-DROP-TABLE PET)
c) (PURGE-FUNCTION #PET)

a) DROP TABLE person;
b) (SQL-DROP-TABLE PERSON)
c) (PURGE-FUNCTION #PERSON)

a) DROP TABLE spieces;
b) (SQL-DROP-TABLE SPIECES)
c) (PURGE-FUNCTION #SPIECES)
```