An algebraic specification of the relational database language SQL

Richard P. van der Luit
Programming Research Group
University of Amsterdam
January 1992
Contents

1. Introduction .................................................................................. 3

2. Formal background ...................................................................... 5
   2.1 Relational Algebra, Relational Databases ......................... 5
   2.2 Database language SQL ..................................................... 9
   2.3 The ASF+SDF formalism ................................................... 14

3. A specification of a relational database system ....................... 20
   3.1 Sample Database .............................................................. 20
   3.2 Modular structure ............................................................ 21
   3.3 Expressions ................................................................. 22
   3.4 Relations, Databases ....................................................... 25
   3.5 Return to the example ...................................................... 27

4. Syntax of SQL ............................................................................ 29
   4.1 Conversion of BNF to SDF ............................................... 29
   4.2 Typechecking preliminaries ............................................. 31
   4.3 Typechecking SQL schema's .......................................... 34
   4.4 Typechecking SQL query specifications ......................... 37

5. Semantics of SQL ...................................................................... 43

6. Conclusions ............................................................................... 45

7. Personal comments about the ASF+SDF system ..................... 47

8. References .................................................................................. 50

Appendices
   A BNF grammar definition of SQL
   B RDBS modules
   C SQL modules
   D SQL typecheck modules
   E SQL to RDBS translation modules
1 Introduction

The subject of this masters thesis is an algebraic specification of the database language SQL.

When faced with the problem of defining a formal (programming) language, we have to investigate the following aspects of this language:

- its concrete (i.e., lexical and context-free) syntax,
- its static semantics, and
- its dynamic semantics.

The first of these topics, the syntax of a language, is usually specified using a context free grammar, like BNF. This describes exactly which clauses belong to the language; all other sentences should be rejected. Low level syntax such as which character set to use, the notion of newline, and definition of layout and comment is given apart from the BNF and specified in natural language.

The next aspect, the static semantics of a language, is meant to define further constraints on the language. A large part of these constraints have normally to do with declarations of objects in the language (variables or routines; identifiers in general) and of their correct use. Especially the declared type of an object is important, therefore checking the static semantics of a given language clause is usually called typechecking and we will use this term throughout the rest of this document. Typechecking rules are written down in a far less formal way than was possible with the syntax. Natural language is usually the appropriate medium.

Finally, the dynamic semantics of a language describes what a given sentence in the language means exactly. Having defined the objects that can be manipulated by the language, the meaning of a clause can be depicted in terms of these objects. The semantics of Pascal, for example, deals with objects like numbers, strings, arrays, records and files. The meaning of a query in a database language SQL will be explained in terms of derived data from databases.

Of these three aspects of a language definition, only the least difficult one is clearly specified using a formal method, the others (including the definition of objects) are usually specified in natural language and are evidently defined less precise.

In this thesis we will investigate if it is practically possible, and if so to which extend, to use a formal method in order to specify syntax, typechecking rules and semantics of the language SQL. SQL is a data definition and manipulation language based on relational databases and has been chosen for a number of reasons:

- SQL is not a traditional programming language, as for example Pascal would have been,
- the language has been defined in an international standard [1] using syntax, syntax rules and general rules, in the way we described above,
- the objects to be defined and manipulated using the language, relational databases, have a clear algebraic notion, and
- many complain that SQL is still incomprehensive in many ways.

The system we used for specifying SQL and its underlying relational databases is the ASF+SDF system [2], which is a environment for the interactive development and implementation of languages. ASF+SDF has been developed in the GIPE project (Generation of Interactive Programming Environments - ESPRIT Project 348).

Provided a (modular) syntax definition of a language, the ASF+SDF system generates interactively a parser and a generic text and structure editor (called GSE) for that language [3]. It then allows to enter some term, which is a sentence in the specified language, and have it parsed on demand of a mouseclick. After the term is parsed without errors, the editor then places its focus on the smallest syntax construction where the mouseclick took place and we are able to inspect the structure of our language definition by moving the focus around.

Changing the syntax definition of the language results in an incrementally changed parser and generated editor, so this system is extremely suitable for developing a language definition interactively.
From a given parse tree the ASF+SDF system also creates an abstract syntax tree, which is used to match with a set of conditional equations. These equations are given in the equations section of a module and form the definition of a term rewriting system with which we can reduce a given term in the language to its normal form. In this way we can, for instance, define how to evaluate an expression, execute a statement or calculate the result of a typecheck function.

In section 2.3 we will present a more elaborate description of the ASF+SDF system, including an example.

The remainder of this paper is organized as follows:

In chapter 2 we present some formal background concerning relational databases, the language SQL and the ASF+SDF formalism.

A specification of a relational database system (RDBS), based on the relational algebra, is given in chapter 3. First, a sample database with some queries is shown to illustrate the kind of system we want. Next, the structure of the syntax modules and each module in turn is described and finally, we return to the sample database to show that we can actually calculate the given queries.

In chapter 4 we deal with the syntax of SQL. First, we describe the conversion of the BNF definition from the standard to an SDF syntax definition. Then, after discussing some design choices, the implementation of a typecheck function for SQL schema's and query specifications is described.

Chapter 5 deals with the semantics of a small but important subset of SQL. This is done by showing specifications of translation functions from SQL language clauses to the RDBS language.

In chapter 6 we reach some conclusions with respect to the objects we had in mind.

Some comments about the practical sides of working with the ASF+SDF formalism and -system are given in chapter 7.

The appendices list the BNF grammar of SQL and all ASF+SDF modules.
2 Formal background

2.1 Relational Algebra, Relational Databases

In this paragraph we present some formal background concerning relations, relational databases and relational algebra in the form of definitions and related issues:

Relations

Let \( D_1, D_2, \ldots, D_n \) be a collection of not necessarily different (finite) sets. Then we can define the product \( D_1 \times D_2 \times \ldots \times D_n \) as the collection of all possible \( n \)-tuples \(<d_1, d_2, \ldots, d_n>\) such that \( d_i \) is an element of \( D_i \) (\( i = 1 \ldots n \)).

\( R \) is called a relation over \( D_1 \times D_2 \times \ldots \times D_n \) when \( R \) is a subset of \( D_1 \times D_2 \times \ldots \times D_n \).

\( D_1, D_2, \ldots, D_n \) are called the domains of \( R \); \( d_1, d_2, \ldots, d_n \) are the attributes of \( R \).

The degree of \( R \) is the number of \( R \)'s attributes (\( n \) in this case); the cardinality of \( R \) is the number of elements (tuples) in \( R \).

A value of \( R \) is a value of an attribute of some tuple in \( R \).

Example 1

Domains

Let Booleans, Integers, Reals and Strings be domains for objects like true, 147, 199.95 and 'Sunday' respectively.

Relations

Students \( \subseteq \) Integers \( \times \) Strings \( \times \) Strings -- i.e. student number, name and phone

Members \( \subseteq \) Integers \( \times \) Strings -- i.e. student number and sport

Contributions \( \subseteq \) Strings \( \times \) Reals -- i.e. sports and price

Instances

Students = {
  < 891247 , 'Al-Hassan', '02990 - 42185'>,
  < 893218 , 'Bakker', '020 - 6123316'>,
  < 896673 , 'Jansen', '020 - 6123316'>,
  < 901274 , 'The', '> }

Members = {
  < 891247 , 'Tennis', '>
  < 893218 , 'Zwemmen', '>
  < 893218 , 'Tennis', '>'
}

Contributions = {
  < 'Zwemmen', 50.00 >,
  < 'Tennis', 75.00 >,
  <'Wandelen', 1.50 >
}

Relational databases

A relational schema is a set of rules which give a description of the defined domains, relations over these domains, and names and domains of attributes, possibly completed with additional constraints on allowed values.

A relational database is an instance of a relational schema.

Example 2

Domains

-- Booleans, Integers, Reals, Strings: predefined
Sports = \{ 'Zwemmen', 'Tennis', 'Wandelen', 'Voetbal' \}
Numbers = \{ n \in \text{Integers} \mid (n \geq 800000) \land (n \leq 919999) \}
-- Note: the first two digits of these numbers represent the year of arrival.

Relations
Students(S\_number: Numbers, name: Strings, phone: Strings)
  Key: S\_number
Members(M\_number: Numbers, M\_sport: Sports)
  Key: M\_number + M\_sport
  \forall M\_number \in \text{Members} \exists S\_number \in \text{Students}: M\_number = S\_number
Contributions(C\_sport: Sports, amount: Reals)
  Key: C\_sport

-- Note: "Key" is a constraint indicating that any occurrence of a value of a key attribute, or
combination of attributes, may not occur more than once in the relation.

Issues related to relational databases

- A relation is an unordered set:
  There is no ordering defined on the domains (which are sets), so there is also no ordering defined
on tuples derived from those domains. In practice however it is often convenient to have such an
ordering, especially for (sorted) listings or other presentations of data.

- Attributes are ordered:
  We cannot reorder the attributes of a tuple at will without loss of meaning, so we have to number
these attributes (first, second, ...). This numbering will in most cases be transparent because of the
use of attribute names as given in the relational schema.

- Relations are not supersets:
  Every tuple from a relation must have a unique appearance in that relation, otherwise a relation
should have been defined as a superset. If one demands us to have unique tuples, one apparently
assumes implicit that e.g. \{a, a\} \neq \{a\}. For several database queries, determining the cardinality of a
relation for instance, it is necessary that duplicate tuples are not removed automatically.

- The key of a relation:
  The key of a relation is defined as a combination of some or all attributes of a tuple. The values of
these attributes must appear unique within the relation. Therefore an instance of a key is
consequently an identification of exactly one tuple (or no tuple at all).

- Constraints upon relations:
  We can restrict the appearance of certain tuples of all possible tuples over \( D_1 \times D_2 \times \ldots \times D_n \)
by specifying field-, intrarecord-, and interrecord constraints. A key of a relation is one form of such
constraint.

- Null-values:
  Each domain can be extended with a omni-typed value null, which stands for unknown,
unavailable, etc. Special logic has to be defined when dealing with the null-value.

- Normalisation:
  A relation R is in first normal form, 1NF, when every value of R is nondecomposable (atomic). There are severl normal forms defined on relations, each next normal form will rule out some sort
of so called anomalies.
  Claim: every relation can be brought to 1NF because of the associativity of \( \times \):
  \[ D_1 \times (D_2 \times D_3) = (D_1 \times D_2) \times D_3 = D_1 \times D_2 \times D_3 \] etc.
Relational algebra

When \( R, R_1, R_2, \ldots, R_n \) are themselves relations, then so are the results of the following expressions:

- \( R_1 + R_2 \) \quad \text{union}
- \( R_1 \ast R_2 \) \quad \text{intersection}
- \( R_1 - R_2 \) \quad \text{difference}
- \( R_1 / R_2 \) \quad \text{division}
- \( R_1 \# R_2 \) \quad \text{product}
- \( (R) \) \quad \text{brackets}
- \( R \mid p(a_1, a_2, \ldots, a_n) \) \quad \text{selection; } p \text{ is a predicate over attribute names of } R
- \( R[b_1, b_2, \ldots, b_m] \) \quad \text{projection; } b_j \text{ is an attribute name of } R
- \( R_1 \# R_2 \# \ldots \# R_x \mid p(a_1, a_2, \ldots, a_y) [b_1, b_2, \ldots, b_2] \) \quad \text{join}

Some of these operations on relations are well-known from the set theory; these are union (normally \( A \cup B \)), intersection (normally \( A \cap B \)), difference (normally also \( A - B \)) and Cartesian product (normally \( A \times B \)). Rather unknown is division: \( A/B \) is the set: values \( x \) such that the pair \(<x,y>\) appears in \( A \) for all values \( y \) appearing in \( B \). Bracketing is used for grouping sub-expressions.

Typical relational operators are selection, projection and join:

Selection is used for selecting a (possibly empty) subset of \( R \) such that for all tuples of the subset the predicate \( p \) holds (and for all tuples which are not selected, the predicate \( p \) does not hold). The predicate \( p \) is taken over the attributes of \( R \), but may also involve other relations as well (then it's a so-called outer-reference).

Example 3
The selection "Students \( \mid S\text{-number} < 900000\)" will list all students (i.e. their number, name and phone) who arrived before 1990. (Remember that the first two digits of the number represent the year of arrival.)

Example 4
The selection "Students \( \mid S\text{-number} \in \{\text{Members} \mid M\text{-number} = S\text{-number}\}" asks for students for who's student-number occurs in Members, i.e. a listing of all students who are enlisted for some sport.

Projection is used for selecting some of the possible attributes resulting from a relation (expression), possibly rearranging the selected attributes.

Example 5
The projection "Students[phone, name]" will list students phone and name.

Two points of interest arrive here:

1. duplicate tuples can be introduced by union, but occur more frequently (and more unexpected) in the result of a projection. As an example: "Which students are sports(wo)men?" can be asked as: "Members[Members]", which results in 4 student-numbers in Example 1, when there is no automatic removal of duplicate tuples and

2. the projection can also serve as a value-selector: not only the needed attributes are listed, but expressions involving those attributes.

The next example will illustrates this last point.
Example 6:
The projection "Employee[surname + lastname, 'will get   $', salary * 1.05]" will give a listing:
    Jan Jaap de Slome will get $105
instead of something less informative like:
    'Slome' 'Jan Jaap de' 100
which will be the result of "Employee[lastname, surname, salary]"

Join is the most important of the relational operators. It is however not a operator on its own, but a
combination of product, selection, and projection.

Example 7:
The query "(Students # Members | S_number = M_number)[name, sport]" lists student names who
practice at least one sport, together with the sport they practice.

This is the most general form of join; most frequently found is the so-called natural join, which is a
combination of two relations sharing the same attribute, and removal of one of these duplicate
attributes, as in "Join(Students, Members, S_number, M_number)[name, phone]".

Issues related to the relational algebra

- Union-compatibility:
  In order to perform "+" "*" and "-" it is necessary that the corresponding attributes of the operands
  are drawn from a common domain.

- Attribute names of a result:
  The product (join) of two different relations can result in attributes with the same name. Sometimes
  even a product of the same relation is wanted and all attribute names will occur twice in the result.
  In such a case, a mechanism for renaming of the attributes is necessary, either by naming the
  relations involved (as in "Students.name") or by giving an alias to relation names. In most database
  systems some ad-hoc mechanism is chosen.

- Priorities and associativity of operators:
  The operators "+", "*" and "#" are associative, "/" and "/" are not associative. Concerning priorities
  there is a preference towards the ordering: "#" > "#" "#" "#" "#". Then, "A # B * C + D" will be
  interpreted as "((A # B) * C) + D". We are not convinced that this is the right ordering.

- Views on relations and databases:
  One mechanism for offering the several users of a database system a different look at the
  information, is by defining views. This is done by declaring a view name and an associated query
  (i.e. a relational expression) over other relations and views. If someone is interested in a list of
  names of all student swimmers for example, one can declare the view:
  Swimmers = (Students # Members | (S_number = M_number) & (sport = 'Zwemmen'))[S_name]
  If, for instance, a new enrollment is made for a student in the relation Members, the view will
  automatically be updated when his or her sport is 'Zwemmen'.

- Cursors (tuple variables for use in relational calculus):
  For procedures on relations, that cannot be done using the relational operations only, one can
  declare a cursor, which is a variable ranging over all tuples of a relation. All relational operators
  are effectively implemented by using some sort of (implicit) cursor; this however may not be visible to
  the user. Therefore some propose that the relational calculus (i.e. a calculus for relations with
  operations expressed in terms of cursor movements and assignments) is a good alternative for the
  relational algebra. It has been shown that both algebra and calculus can express exactly the same, so
  they can be considered equally powerful.
2.2 Database language SQL

The name SQL stands for Structured Query Language, and was designed in the mid and late 1970s by IBM, based on their System R technology. SQL has facilities for defining, manipulating, and controlling data in a relational database. There are numerous commercially SQL-based products; the two most important are ORACLE by Oracle Corporation and SQL/DS by IBM. In 1986 SQL became an official international standard by ANSI and ISO.

The complete definition of the SQL standard is written in the Final draft ISO 9075-1987(E) Database Language SQL (Nov. 1986) [1]. Except for some preliminary overview and concepts, this definition is given by specifying the several syntactic elements of SQL, in terms of:

- Function: a short statement of the purpose of the element,
- Format: an extended BNF definition of the syntax of the element,
- Syntax Rules: additional syntactic constraints that the element shall satisfy, and
- General Rules: a specification of the run-time effect of the element.

The BNF definition of SQL is listed in Appendix A for reference and has the following extensions to standard BNF notations:

- []  optional element
- ...  elements repeated one or more times
- {}  grouping of sequences of elements

A short abbreviated definition of <schema> (§6) and <query specification> (§5), together with some explanation will be presented hereafter. Our extensions to standard BNF are:

- Item*  zero or more repetitions of item
- Item+  one or more repetitions of item
- [Item ","]*  list of zero or more items, separated by ","
- [Item ","]+  list of one or more items, separated by ","
- [Item]  optional item
- (Item)  grouping of item

A relational schema is defined in SQL as:

```
Environment ::=  
    CREATE ENVIRONMENT Ident Schema*

Schema ::=  
    CREATE SCHEMA AUTHORIZATION Ident SchemaElement+

SchemaElement ::=  
    CREATE TABLE TableName "(" {ColumnDef | TableConstraint}" , ")
| CREATE VIEW TableName ["(" {Ident ","}+ ")"]
| CREATE VIEW TableName ["(" {Ident ","}+ ")"]
| AS QuerySpec [WITH CHECK OPTION]
| GRANT (ALL PRIVILEGES | {Action ","}+ ) ON TableName
| TO { (PUBLIC | Ident ) ","}+ [WITH CHECK OPTION]

TableName ::= [Ident ","] Ident

TableConstraint ::=  
    (UNIQUE | PRIMARY KEY) "(" {Ident ","}+ ")"
| FOREIGN KEY "(" {Ident ","}+ ")"
| REFERENCES TableName ["(" {Ident ","}+ ")"]
| CHECK "(" SearchCondition ")"

Action ::=  
    SELECT
| INSERT
| DELETE
```
UPDATE ["(" {Ident ","}+ ")" ]
REFERENCES ["(" {Ident ","}+ ")" ]

An environment is not part of the SQL language, but an 'implementor-defined concept' as defined in §6.1 <schema> Syntax Rules 1. A schema lists an authorization identifier and one or more table-, view- or privilege definitions.

A table definition (CREATE TABLE ...) defines a table (i.e. a relation in first normal form) with columns (attributes) defined in the column definitions. Since two tables in different schemas can have the same name, such a table name may be preceded by the identifier of the enclosing schema, as in "SCHEMA_IDENT.TABLE_IDENT". Table constraints define the constraining of allowed rows (tuples) to be inserted in a table.

A view definition (CREATE VIEW ...) is the method to create a specific look at the data, as has been discussed in the introduction. It defines the name of the view, an optional list of column names (a view can be seen as a table with rows too) and a query resulting in the desired data.

Operations requested by a user of the database can be limited by the creator of the database by (not) giving allowance to specific users to perform certain operations (called privileges) on a defined table. This is done using a privilege definition (GRANT ...).

ColumnDef ::=  
  Ident DataType [DEFAULT (Literal | USER | NULL)] ColumnConstraint*  

DataType ::=  
  CHARACTER ["(" Unsigned ")" ]  
  | CHAR ["(" Unsigned ")" ]  
  | (NUMERIC | DECIMAL | DEC) ["(" Unsigned ["," Unsigned] ")" ]  
  | INTEGER  
  | INT  
  | SMALLINT  
  | FLOAT ["(" Unsigned ")" ]  
  | REAL  
  | DOUBLE PRECISION  

ColumnConstraint ::=  
  NOT NULL [UNIQUE | PRIMARY KEY]  
  | REFERENCES TableName  
  | CHECK "("-SearchCondition ")"  

A column definition is specified as an identifier (attribute name), a datatype (only a limited set of datatypes exist in SQL and there is no mechanism to define further domains), an optional default (i.e. a value to be inserted when no specific value is given) and a set of constraints on the allowed values of the column.

Example 8
CREATE SCHEMA AUTHORIZATION ADMINISTRATION
CREATE TABLE SPORTS (  
  SPORT CHAR(15)  
)  
CREATE TABLE STUDENTS (  
  NUMBER NUMERIC(6)  
  NAME CHAR(25)  
  PHONE CHAR(15)  
)  
CREATE TABLE MEMBERS (  
  NUMBER NUMERIC(6)  
  SPORT CHAR(15)  
)
The most important information retrieval function in SQL is a query specification (query for short). Query specifications occur in many places in SQL, but their most common use is an interactive query typed by a user of the database.

QuerySpec ::=  
   SELECT [ALL | DISTINCT] ("*" | [ValueExpr ","{"}]+)  
   FROM { (TableName [Ident]) ","{"}+  
   [WHERE SearchCondition]  
   [GROUP BY { (ColumnSpec) ","{"}+}]}  
   [HAVING SearchCondition]  

ColumnSpec ::=  
   [TableName ",." | Ident  

SearchCondition ::=  
   SearchCondition AND SearchCondition  
   | SearchCondition OR SearchCondition  
   | NOT SearchCondition  
   | Predicate  
   | "(" SearchCondition ")"  

Predicate ::=  
   ValueExpr ("=" | ">" | "<" | ">=" | "<=" | ">=") (ValueExpr | SubQuery)  
   | ValueExpr [NOT] BETWEEN ValueExpr AND ValueExpr  
   | ValueExpr [NOT] IN (SubQuery | {ValueSpec ","{"}+})  
   | ColumnSpec [NOT] LIKE ValueSpec [ESCAPE ValueSpec]  
   | ColumnSpec IS [NOT] NULL  
   | ValueExpr ("=" | ">" | "<" | ">=" | "<=" | ">=") (ALL | SOME | ANY) SubQuery  
   | EXISTS SubQuery  

SubQuery ::=  
   "(" SELECT [ALL | DISTINCT] ("*" | ValueExpr)  
   FROM { (TableName [Ident]) ","{"}+  
   [WHERE SearchCondition]  
   [GROUP BY { (TableName) ","{"}+}]}  
   [HAVING SearchCondition]  

A query specification has in general the following outline:

   SELECT V1, V2, ..., Vn  
   FROM T1, T2, ..., Tm  
   WHERE SearchCondition  
   GROUP BY C1, C2, ..., Cx  
   HAVING SearchCondition

It combines Cartesian Product (FROM T1, T2, ...), selection (WHERE SearchCondition) and extended projection (SELECT V1, V2, ...).

A query without projection is written down as "SELECT * FROM ... WHERE ...."

The result of a query can be explained as follows:
1. The Cartesian Product of all tables mentioned in "FROM T1, T2, ..., Tm" is calculated.  
   (As with the definition of tables in a schema, the name of a table may be qualified with the  
   corresponding authorization identifier of the schema. In case of ambiguous table names this  
   qualifier is mandatory. Other ambiguities, e.g. when a product of a table with itself is required, are  
   resolved by specifying a correlation name, i.e. an additional identifier after the table name.)

2. Each row of this product is inspected with respect to the optional search condition. This is a  
   boolean condition built out of basic predicates involving some or all attributes of the row. The  
   result consists of all rows for which this search condition holds.

3. The former result is rearranged to form groups with equal values for the columns mentioned in the  
   optional GROUP BY clause.  
   (A column is referenced using a single identifier, the column name. In case of ambiguity it is  
   preceded with the name of a table, which can be an original table name or a correlation name.)

4. Each resulting group is in turn tested against the search condition of an optional HAVING clause.  
   The result is formed out of all groups for which this condition holds.

5. The result of 1-4 is further processed by the select list "V1, V2, ..., Vn".  
   "SELECT *" yields all attributes just as they are. "SELECT V1, V2, ..., Vn" results in a table with  
   degree n, each attribute value calculated from the value expressions V1..Vn (see below).  
   The keyword DISTINCT will remove all duplicate tuples in the final result.

Example 9
Select candidates for the swimming competition...:
SELECT NAME, PHONE
FROM STUDENTS, MEMBERS
WHERE (STUDENTS.NUMBER = MEMBERS.NUMBER)
   AND (SPORT = "Zwemmen")

Example 10
Print student names which are enrolled for more than one sport...:
SELECT NAME
FROM STUDENTS, MEMBERS
WHERE STUDENTS.NUMBER = MEMBERS.NUMBER
GROUP BY NAME
HAVING COUNT(SPORT) > 1

The possible predicates in a search condition are briefly illustrated with the following examples.  
(Note: many predicates allow an optional NOT to be specified, the result of the predicate is negated  
then.)

Examples 11
• value comparison:
   SELECT * FROM STUDENTS WHERE STUDENTS.NUMBER = 901274
   (Possible comparisons are also <=, <, >, <=, and >=. A subquery is allowed on the right hand side  
   of the comparison, it should then result in a single value.)

• value within a range of values:
   SELECT * FROM STUDENTS WHERE NUMBER BETWEEN 900000 AND 909999
   (i.e. 900000 <= NUMBER <= 909999)

• value occurrence in a set of values:
   SELECT NUMBER FROM MEMBERS WHERE SPORT IN 'Tennis', 'Voetbal'
(Here too a subquery is allowed on the right hand side, in this case it should result in a single column.)

- synonym comparison: (all names beginning with letter C)
  
  SELECT NAME FROM STUDENTS WHERE NAME LIKE 'C%'

- test if column has the null value:
  
  SELECT SPORTS FROM CONTRIBUTIONS WHERE AMOUNT IS NULL

- universal quantifier: (select oldest students)
  
  SELECT * FROM STUDENTS
  WHERE NUMBER <= ALL (SELECT NUMBER FROM STUDENTS)

- existence quantifier: (select non-sports(wo)men)
  
  SELECT * FROM STUDENTS
  WHERE NUMBER <> ANY (SELECT NUMBER FROM MEMBERS)

(SOME and ANY are identical)

- existence (also illustration of outer reference and of use of a correlation name):
  
  SELECT NAME FROM STUDENTS
  WHERE EXISTS
  (SELECT NUMBER FROM MEMBERS SPORTERS
   WHERE STUDENTS.NUMBER = SPORTERS.NUMBER)

Value expressions are common expressions resulting in a single value, as opposed to table expressions resulting in tables. These expressions are composed of literal values (string, numeric, or the special keyword USER), references (to columns, parameters, or variables), and results of functions. They are bound together with operators "+", "-", "*", and "/". The associativity- and priority properties of the operators are forced by the original BNF definition and are not visible here.

ValueExpr ::= 
  ValueExpr "+" ValueExpr |
  ValueExpr "-" ValueExpr |
  ValueExpr "+" ValueExpr |
  ValueExpr "/" ValueExpr |
  "+" ValueExpr |
  "-" ValueExpr |
  ValueSpec |
  ColumnSpec |
  SetFuncSpec |
  "(" ValueExpr ")"

ValueSpec ::= 
  Ident [[INDICATOR] Ident] -- parameter
  | ":" Ident [[INDICATOR] ":" Ident] -- variable
  | Literal

Literal ::= 
  "" -- string
  | ["+" | "]" ] Unsigned
  | ["=" | "]" ] Unsigned
  | ["+" | "]" ] Unsigned "." Unsigned
  | Exact "E" ["+" | "]" ] Unsigned
  | USER

SetFuncSpec ::= 

- 13 -
COUNT "(" "*" ")"
| (AVG | MAX | MIN | SUM | COUNT) "(" DISTINCT ColumnSpec ")"
| (AVG | MAX | MIN | SUM) "(" [ALL] ValueExpr ")"

Using the built-in functions one can count the number of rows, sum the values of a column, or
determine the minimum, maximum, or average of a column.

Examples 12
• determine the number of students:
  
  SELECT COUNT(*) FROM STUDENTS

• find oldest student number:
  
  SELECT MIN(NUMBER) FROM STUDENTS

• find the name of oldest students:
  
  SELECT NAME FROM STUDENTS WHERE NUMBER = MIN(NUMBER)

With the exception of GROUP BY, built-in functions cannot be intermixed with attribute names;

  SELECT NAME, COUNT(*) FROM STUDENTS

, for instance, is not allowed.

2.3 The ASF+SDF formalism

In this section we will demonstrate the practical use of the ASF+SDF formalism. For a more formal
description, see [2]. Theoretical background about algebraic specifications and term rewriting
systems can be found in [4].

So without going into much detail, we will explain the modular structure of ASF+SDF
specifications, the various sections they consist of, and the parsing and reduction of terms.

An ASF+SDF specification is composed of a number of modules in which we define the syntax of a
language. Such a syntax definition consists of sort names, lexical syntax, and context-free syntax and
can either be exported (i.e. visible to the outside world) or hidden. Modules can import other
modules and so use their exported syntax. The modules of a specification are bound together by such
import relations. Next, we show two modules forming the specification of the boolean datatype. The
first module, Layout, does not import any other modules, but is instead a basic module to define the
predefined sort LAYOUT. This definition of blank space or comment is used by the scanner to skip
when such phrases are encountered in the input.

module Layout
exports
  lexical syntax
  [ \t\n]"%"~[\n]* [\n] -> LAYOUT
end Layout

The first definition specifies layout to be a space character, a tab or the newline character; the brackets
around them indicate a set of possible characters. The second definition tells us that everything
between two percent signs and the newline should also be treated as blank space, so this is our
definition of comment. (Note: the phrases "module Layout" and "end Layout" are not part of the
ASF+SDF implementation, but included here for readability.)

Our next module, Booleans, first imports module Layout we defined just before and then defines the
datatype booleans with sortname BOOL. Booleans can have values "true" or "false" and can be
manipulated with operators "not", "or" and "and".
module Booleans
imports Layout
exports sorts
  BOOL
lexical syntax
"true" -> BOOL
"false" -> BOOL
context-free syntax
not BOOL -> BOOL
BOOL or BOOL -> BOOL { left }
BOOL and BOOL -> BOOL { left }
"(" BOOL ")" -> BOOL { bracket }
variables
"bool" [0-9]* -> BOOL
priorities
not BOOL -> BOOL >
{ left: BOOL and BOOL -> BOOL, BOOL or BOOL -> BOOL }
equations
[not1] nct true = false
[not2] nct false = true
[or1 ] true or bool = true
[or2 ] false or bool = bool
[or3 ] bool or true = true
[or4 ] bool or false = bool
[and1] true and bool = bool
[and2] false and bool = false
[and3] bool and true = bool
[and4] bool and false = false
end Booleans

This module defines and exports the sortname BOOL, lexical syntax for BOOL-constants, and context-free syntax for the prefix unary "not" and infix binary "and" and "or" operators, and for bracketing subexpressions. The separation between the lexical syntax part and the context-free syntax part gives the distinction of what is recognized by the scanner and what is recognized by the parser. Literals occurring in context-free syntax definitions are transferred to the scanner as well, so we could have defined the boolean constants in this section also.

With the syntax just made, we can enter terms like "true", "false", "true or false" etc. But what happens to a term like "true or false or true"; should this be treated as "(true or false) or true" or as "true or (false or true)? This ambiguity problem is solved by the associativity attribute left attached to the functions "and" and "or", so the expression mentioned is parsed as "(true or false) or true". Other possible associativity attributes are right, assoc and non-assoc.

In the priorities section we stated that the relative priority of "not" to be higher than the priority of "and" and "or". Therefore an expression like "not true and false" is interpreted as "(not true) and false" (yielding false) and not as "not (true and false)" (yielding true).
Ambiguity problems which arise from intermixed use of "and" and "or" are solved by the group associativity attribute left in the priorities section, stating that "true and false or true" should be parsed as "(true and false) or true". Other possible attributes are: right and non-assoc.

The context-free function "(" BOOL ")" -> BOOL is meant to group boolean expressions in order to circumvent the associativities and priorities specified. The attribute bracket is attached to it, so this function will not appear in the derived abstract syntax for the boolean language.

- 15 -
We also declared an infinite set of boolean variables in the appropriate section of the exported language. The names of these variables can be used in the equations and follow the given syntax, so `bool, bool1, bool789, bool'. bool7'` are all legal boolean variables.

The equations specify the behaviour of booleans in a derived term rewriting system. They follow the normal rules for truth tables. After having read our specification into the system, we can enter a term over the boolean language, for example "true or not true and false". This term is syntactically correct and can thus be parsed by clicking the mouse somewhere in the term, e.g. on the second "true". Then we get a focus on this subterm and we can inspect our language clause by moving this focus around. The following in an example showing the repeated zooming out of the focus (indicated by outlined character format):

```
  true or not true and false
  true or not true and false
  true or not true and false
```

Focus movements are based on the traversal movements in a syntax tree: zoom in (down to first subnode), zoom out (one node up), next child (to the right on same level), previous child (to the left on same level).

The modification of modules (syntax or equations) and the editing of a term, including focus movement, is done by using a generated text and structure editor GSE, see [3]. Such an editor is generated from a syntax definition: the syntax for booleans generates GSE instances specially tailored to the boolean language. Editing SDF modules follows the same principle: an editor is generated based on the syntax definition of an SDF module (either syntax- or equation modules).

Beside normal edit functions and focus positioning, these editors allow us to build and/or inspect terms in the specified language by means of the 'expand' menu. Open terms in our language may contain so called meta-variables (sortnames surrounded by '<' and '>') which can be replaced with one of the options from the expand menu. We will build the term from above using this feature:

```
<BOCOL>
<BOOL> and <BOOL>
<BOOL> and <BOOL>
<BOOL> or <BOOL> and <BOOL>
<BOOL> or <BOOL> and <BOOL>
true or <BOOL> and <BOOL>
true or <3COL> and <BOOL>
true or not <BOOL> and <BOOL>
true or not <BOOL> and <BOOL>
true or not true and <BOOL>
true or not true and <BOOL>
true or not true and false
```

(expand: BOOL and BCOL -> BOOL)
(mouseclick)
(expand: BOOL or BOOL -> BOOL)
(mouseclick)
(expand: BOOL; typing "true")
(mouseclick)
(expand: not BOOL -> BOOL)
(mouseclick)
(expand: BOOL; typing "true")
(mouseclick)
(expand: BOOL; typing "false")

In conjunction with a closed term, the expand menu can be used to determine the sort of a subterm; this feature is especially handy when we edit lists or when we have terms over subsorts (by injection of one sort into another).

Now we have a well formed term in the boolean language, we can reduce this term to its normal form in the corresponding TRS by rewriting it repeatedly using the equations given. Our term will be reduced as follows:
true or not true and false (rule [not1])
true or false and false (rule [or1] or [or4])
true and false (rule [and1] or [and4])
false (rule [not1])

To give a more elaborate example of a specification in ASF+SDF, we will add another module which specifies a datatype sets. A set is defined as a collection of zero or more elements between curly braces, separated by commas, such as in: {Monday, Tuesday, Wednesday, Thursday, Friday}. Elements of a set are defined as either a capital letter followed by letters or digits, or one or more digits. Defined operations on sets are union ("+")", difference ("-"), and intersection ("*"). We also defined a grouping function and a set-membership predicate "in".

module Sets
imports
  Layout
  Booleans
exports
  sorts
    ELEM SET
lexical syntax
  [A-Z][A-Za-z0-9]* -> ELEM
  [0-9]+ -> ELEM
context-free syntax
  "{" {ELEM },"} " -> SET
  SET "+" SET -> SET { right }
  SET "-" SET -> SET { right }
  SET "*" SET -> SET { right }
  "(" SET ")" -> SET { bracket }
  ELEM "in" SET -> BOOL
variables
  "elem" [0-9]* -> ELEM
  "set" [0-9]* -> SET
hiddens
  variables
    "e"[0-9]* -> ELEM
    "e*"[0-9]* -> {ELEM }""*"+
    "e+"[0-9]* -> {ELEM }""+
  equations
[ norm ] {e*1, e, e*2, e, e*3} = {e*1, e, e*2, e*3}
[ union ] {e*1} + {e*2} = {e*1, e*2}
[ diff1 ] { } - {e*} = {}
[ diff2 ] {e, e*1} - {e*2} = {e} + {{e*1} - {e*2}}
  when e in {e*2} != true
[ diff3 ] {e, e*1} - {e*2} = {e*1} - {e*2}
  when o in {e*2} = true
[ intr1 ] { } * {e*} = {}
[ intr2 ] {e, e*1} * {e*2} = {e} + {{e*1} * {e*2}}
  when e in {e*2} = true
[ intr3 ] {e, e*1} * {e*2} = {e*1} * {e*2}
  when e in {e*2} != true
[ elem1 ] el in { } = false
[ elem2 ] el in {e2, e*2} = true
  when el = e2
[ elem3 ] el in {e2, e*2} = el in {e*2}
  when el != e2
end Sets
A feature not yet discussed in the SDF formalism is the definition and treatment of lists. In general a list structure is defined as: \{<sortname> <separator>\} where <separator> is a terminal and \# is either "*" or "+". This denotes zero or more resp. one or more repetitions of subterms of the proper sort, separated by <separator>. Without <separator> we also may write <sortname>\#. Also variables may be of such a list sort, see for example the declaration of variables "e*" and "e+". We declared these in the hidden section of module Sets, so they will not appear in the exported syntax.

Another feature, conditional equations, can be found in several equations of module Sets. The general syntax of an conditional equation is either

\[... \# t_1 = t'_1, t_2 = t'_2, ..., t_n = t'_n \Rightarrow t = t'\]

or

\[... t = t' \text{ when } t_1 = t'_1, t_2 = t'_2, ..., t_n = t'_n.\]

Both statements that we rewrite t to t' after we proved to be able to rewrite ti to ti' for i = 1..n.

The first equation of module Sets, [norm], will remove duplicate occurrences of an element in a set. A union of two sets is easily defined as a listing of all elements of both sets, provided that the former rule removes duplicates. As an example we get the following reduction:

\[\{\text{red, yellow, blue}\} + \{\text{red, white, blue}\} \Rightarrow \{\text{red, yellow, blue, red, white, blue}\} \Rightarrow \{\text{red, yellow, blue, white, blue}\} \Rightarrow \{\text{red, yellow, blue, white}\}\]

Testing for duplicate elements each time a set is manipulated in the rewriting process, is quite inefficient. Therefore we comment out the normalization rule and introduce an explicit unique operator:

"unique" SET -> SET
\%
[norm ] \{e*1, e, e*2, e, e*3\} = \{e*1, e, e*2, e*3\}
[uniq1] unique \{ \} = \{ \}
[uniq2] unique \{e,e\} = \{e\} + unique (\{e\} - \{e\})

The operators "." and "#" are defined in terms of the set-membership predicate "in". The definition of this boolean function has a nasty property: it is based on (in)equality of terms, more precise it is based on (in)equality of normal forms of terms of sort ELEM. If we want elements like 1, 736, 004, etc. to be treated as numbers, we will not get the right membership function. Since leading zero's will not be removed, 004 does not equal 4 and so "4 in \{004\}" will yield false. One solution is to make sure that all ELEM's have one unique normal form:

variables "Chars" -> CHAR+

[norm] \text{elem("0" Chars) = elem(Chars)}

We defined a variable "Chars" to be one or more occurrences of the predefined sort CHAR. Next, we added an equation which makes sure that leading zero's of a term of sort ELEM are automatically removed. The function elem(CHAR+) -> ELEM is provided by the system for all lexical sorts. These functions allow us to manipulate lexical objects at a low level. A sample reduction will be:

\[004 \text{ in } \{1, 2, 3, 4, 5, 6, 7, 8\} \Rightarrow 04 \text{ in } \{1, 2, 3, 4, 5, 6, 7, 8\} \Rightarrow 4 \text{ in } \{1, 2, 3, 4, 5, 6, 7, 8\} \Rightarrow 4 \text{ in } \{2, 3, 4, 5, 6, 7, 8\} \Rightarrow 4 \text{ in } \{3, 4, 5, 6, 7, 8\} \Rightarrow 4 \text{ in } \{4, 5, 6, 7, 8\} \Rightarrow \text{true}\]

In practise, we will have some basic datatypes T at our disposal, with at least the equality function defined on them (T "=" T -> BOOL), and then define a constant set as "\{ "T", "," \} " -> T_SET.
An even better suggestion, not supported by the current ASF+SDF implementation, is the possibility to parameterize module Sets with sort ELEM and the equality function. Binding the parameters ELEM and "=" with an actual sortname of a datatype and an equality function, results in a proper set definition of that datatype.

We have not added a priorities section to our specification of sets, but we could have done this as follows, giving the priorities we suggested in section 2.1:

```
priorities
    "#" > "*" > {right: "+", "-"}
```

which states that A # B * C + D - E is treated as if we had written ((A # B) * C) + (D - E).
3 A specification of a relational database system.

In this chapter we shall describe a specification for a relational database system, called RDBS. We have given a full listing of the work actually done in the appendices, here we describe how the work was done and why it has been done that way.

Serving as an illustration for the kind of system we want, resembling much the examples given in the introduction, we present a sample database and some operations on them in §3.1. We will revert to this example in many cases in the rest of this document.

Then, we describe the modular structure in §3.2 and more about the details of expressions and relations in §3.3 and §3.4 respectively.

Finally, in §3.5 we show the result of evaluating our sample database in §3.6.

3.1 Sample database

The following serves as our sample database; it consists of a schema, database and a program:

schema
Sports == {'Zwemmen', 'Tennis', 'Wandelen', 'Voetbal'}
Numbers <= {n @ Integers | (n >= 800000) & (n <= 919999)}
Amounts <= {a @ Reals | (amount >= 0.0) & (amount <= 300.0)}
Students <= {<number, name, phone, sport> @ Numbers # Strings # Strings}
Members <= {<number, sport> @ Numbers # Sports}
Contributions <= {<sport, amount> @ Sports # Amounts}
Account <= {<name, phone, amount> @ Strings # Strings # Amounts}
Swimmers <= {<name> <= <number, name, phone> @ Students}
database
Sports = {'Zwemmen', 'Tennis', 'Wandelen', 'Voetbal'}
Students = {
  <891247, 'Al-Hassan', '02990 - 42185'>,
  <895218, 'Bakker', '020 - 6123316'>,
  <896673, 'Jansen', '020 - 6123316'>,
  <901274, 'The', '>'}
Members = {
  <891247, 'Tennis'>,
  <895218, 'Zwemmen'>,
  <895218, 'Tennis'>}
Contributions = {
  <'Zwemmen', 50.00>,
  <'Tennis', 75.00>,
  <'Wandelen', 1.50>}
Swimmers = {<name> <= <n1,name,p,n2,sport> @ Students # Members |
  (n1 = n2) & (sport == 'Zwemmen')}
Account = { }
X = { }
Y = { }
program
Students <= Students + {<910001, 'Nieuwland', '015 - 147203'>};
Students <= Students + {<number, name, phone> @ Students | number = 901274};
Contributions <= {<sport,amount * 1.10> <= <sport,amount> @ Contributions};
Account <= {<number, name, amount> <=
  <number, name, phone, number2, sport1, sport2, amount> @ Students # Members # Contributions |
  (number = number2) & (sport1 = sport2)};
X <= Swimmers;
Members <= Members + {<910001, 'Zwemmen'>};
Y <= Swimmers;
end
The schema-part of this example is used to define the relations, their attribute names, and corresponding domains. The domains Booleans, Integers, Reals and Strings are some predefined datatypes. Sports, Numbers and Amounts are typical examples of user-defined domains. Students, Members, and Contributions are the relations in which we really put data in the database (base relations). Account and Swimmers are relations intended to gather derived information from the base relations. Swimmers is an example of a view which keeps track of student names who are enlisted for the swimming course.

The database-part lists the actual contents of the relations, i.e. the base relations and the temporary relations X and Y. Here we see the actual definition of the view Swimmers: a query on the product of Students and Members.

The program following the schema and database contains a list of zero or more statements which are assignments of the form: RelationName "<-" RelationalExpression "."
The first of these statements is a special case of union namely the addition of a new student to the Students relation. The next one makes use of the assumption that each student has a unique student number, so this statement removes a student from Students. The increase of contribution with 10% is an example of a database update by using the extended projection. A conditional update, e.g. an increase of only some contributions, although it looks a little bit laborious, can be expressed as well.

The program continues with a query of what amounts have to be paid and by whom, after the update of the Contributions. The last three statements show the use of a view: after a student is enrolled or deregistered from some sport by insertion or deletion of a relevant entry in the Members relation, the use of Swimmers is automatically updated with this change.

### 3.2 Modular structure

Our specification of a relational database system has been divided over several modules. The import graph representing the import of modules in one another is given in Figure 1 in which M1 --> M2 means: M1 is imported in M2 or, equivalently, M2 imports M1:

![Import graph of RDBS modules](image)

**Figure 1:** Import graph of RDBS modules.
The import relations shown are the explicit imports as can be found in the imports section of the modules. Implicit imports, like Chars in Values via Strings, are not shown since we actually did not want the signature of Chars appearing in Values (SDF has not yet a mechanism for restricting exported syntax). We could have left out the import of Boolean in Values, because this is implicitly done via the import of e.g. Reals in Values; but leaving out Reals from Values may not result in loosing Boolean syntax in Values!

Not shown in Figure 1 is the general occurring import of modules Layout and Lisp.

Layout defines the necessary layout characters space, tab and newline and the notion of comment (either two percent signs initiate comment till end of the line or all text between two percent signs is considered as comment). This module also exports some CHAR variables, which are needed for several low-level functions which are automatically provided by the system for all lexical definitions.

Lisp defines relevant syntax for hybrid implementations [9], like we have done with Reals and Chars [see §3.3].

The general ideas behind this modular structure are as follows:

1. the availability of a library of (predefined) datatypes like Booleans, Integers, Reals, and Strings with the possibility to extend these with newly needed datatypes,
2. the combination of values (belonging to some datatype) with variables to form expressions,
3. the binding of variables in an expression, i.e. replacing such a variable with an actual value, and calculation of the result,
4. a specification for a datatype tuple (i.e. a row of objects), preferable parameterized with the objects to be placed in these tuples,
5. definition of relations and implementation of all (set- or relational-) operators, and
6. definition and actual implementation of database assignments in a schema-database-program.

These issues return also in the next two paragraphs.

3.3 Expressions

In this section we describe the underlying datatypes, expressions over these datatypes and calculation of the results by binding values to variables. We will present the relevant modules bottom-up: 'higher' modules import and use the syntax defined in 'lower' modules.

First, we define the needed datatypes for our database system (booleans, integers, reals and strings), preferably each in its own module. The definitions include sortnames, constants (generally in the lexical section), operations (with their associativity and priorities properties), some predicates (at least equality and inequality, and other comparisons when an ordering can be defined) and variables of the newly defined sorts. These separate modules, possibly from a predefined library, together with a definition of variables can then be combined to form the kind of expressions we want.

Note incidentally the dual meaning of the symbol "=" in e.g. module Booleans. We would like to express the fact that two booleans are considered equal when they have the same normal form:

\[ b_1 = b_2 \implies b_1 = b_2 \]

Only because we can surround \( b_1 = b_2 \) with the bracket-function (BOOL) \( \rightarrow \) BOOL, the ambiguity of \( b_1 = b_2 \) is true can be resolved without user-interaction.

Module Integers specifies the arithmetic operations additions, subtraction and multiplication and the greater-than predicate using a lookup function LIST "." DIGIT \( \rightarrow \) INT, which speeds up the calculations considerably (\( O(\log n) \) instead of \( O(n) \)). For implementation of datatype real in module Reals, we decided to use the hybrid feature of ASF+SDF. Main consequence thereof is that REAL's follow the lexical syntax of reals in Lisp. Note that in both Integers and Reals extra equations are needed for dealing with the negative cases (- INT and - REAL).
Module Chars is just there to provide some comparison predicates on single characters as a basis to
define comparison on strings. Since the alternative is to write out at least 256 equations like \(\text{ord('A')} = 65\), we again made use of hybrid implementation, namely for a hidden function \(\text{ord(CHR)} \rightarrow \text{INT}\) (note that sortname CHAR is unfortunately already taken). By the way, a CHR is lexically defined as
some character, followed by a tilde \(\sim\) and not as a character between single quotes to avoid the
ambiguity with a string-constant of just one character.

Module Strings implements strings (also an auxiliary function \(\text{length(STR)} \rightarrow \text{INT}\) and the operation
\(\text{INT} \times \text{STR} \rightarrow \text{STR}\), i.e. \(80 \times =\) will give a nice page separator).

Having 'all' the datatypes we need, we can combine them into a single sort VAL (stands for 'value')
by means of injection. This is done in module Values, which also defines some auxiliary functions
involving intermixed sorts. The special value "null" is used to denote the unknown or unavailable
value as mentioned in the introduction.

Arrived here we can write down expressions like true and false = true, 'hello' + 'there' or \(16 + 48\)
but not expressions like 'hello' + '48' or false = 0.

In order to be able to perform queries on our database, we need variables which can hold a value of
some datatype. So first we define the syntax for variables: as usual these variables follow the
syntactic rules of an identifier, as can be seen in module Variables.

The easiest way to incorporate variables into expressions is to inject the sort VAR into the subsequent
datatype sorts (e.g. \(\text{VAR} \rightarrow \text{STRING} \lor \text{VAR} \rightarrow \text{VAL}\)). Then we can unambiguously write down
\(100 - a\) but not \(100 + 4\). Expressions like \(a + b\) however result in \(2n + 1\) ambiguities when \(n\) is the
number of distinct datatypes. Later on, after binding values to variables, the expressions are
evaluated correctly, that is \(100 + 4\) reduces to \(104\) and \(100 + 4\) will not occur.

After using the interactive disambiguator extensively, we decided to adopt another method. Another
reason for this is that we could not write down an expression like \(100.0 + 4\) since the function REAL
"+" \(\text{INT} \rightarrow \text{REAL}\) does not exist. Although we can provide such a function, or force the user to use
typecast functions, the following method is more elegant.

Module Expressions defines the sort EXPR, which includes variables, the basic datatypes and all the
operators we want for expressions. Valid expressions are \(a = b, 4 + \text{'zip'}\) and true & false. Note that
now the last of these expressions is ambiguous: either \(\text{BOOL} \& \text{BOOL} \rightarrow \text{BOOL}\) or \(\text{EXPR} \& \text{EXPR}
\rightarrow \text{EXPR}\). At least the following methods to solve this ambiguity exist:

1. rename all the overloaded functions on 'lower' datatypes, like renaming +: \(\text{INT} \# \text{INT} \rightarrow \text{INT}\)
to plus(\(\text{INT} \rightarrow \text{INT}\)), but this would contradict our idea to define datatypes separately in a
library,
2. the same renamings using a renaming mechanism in the specification (does not exist in ASF +
SDF),
3. use the same priorities to force the system to choose among the possibilities, e.g. "\(\text{INT} \rightarrow \text{INT}"
\(\text{EXPR} \& \text{EXPR} \rightarrow \text{EXPR}\) or v.v. (not possible in ASF + SDF this way), or
4. hiding of the automatic export of the 'lower' functions to the outside world (also not implemented
in ASF+SDF; all imported syntax is also exported automatically).

Since expressions only involving constants, like true & false or \(4 + 17 > 12\) will occur only in rare
situations (false resp. \(21 > 12\) or simply true will suffice), we have let these ambiguities for what
they are.

Evaluating an expression is done by binding variables in the expressions, i.e. replacing each
occurrence of a variable with an actual instance (value) of that variable. Since we hold values,
variables and expressions in lists (tuples, rows), we refer below to module Tuples for this binding
mechanism. The result of a binding is an expression without variables, so only involving constants
of one or several datatypes.
Example 1
Suppose we have two variables a and b with values 4 and 6 respectively. Replacing the values for the variables in the expression a + b >= 10 results in 4 + 6 >= 10 and is then evaluated as 10 >= 10, yielding true.

Evaluation of these 'closed' expressions is done by the equations of module Expressions. The following notes apply to these equations:

- Functions like sort(SORT) -> SORT, e.g. bool(BOOL) -> BOOL, had to be provided for forcing the type of the argument; see for instance the equation
  \[ _{\text{Expr1}} + _{\text{Expr2}} = \text{int}(\_\text{Int1} + \_\text{Int2}) \text{ when } _{\text{Expr1}} = \_\text{Int1}, _{\text{Expr2}} = \_\text{Int2}. \]
  Via this typecasting the function "+" is 'forced' to be integer addition instead of expression addition.

- For each expressions function all combinations of all datatypes had to be considered meaningful; although REAL "+" INT -> REAL (and v.v.) was not provided by the datatypes we can simulate this function as has been done in the equation
  \[ _{\text{Expr1}} + _{\text{Expr2}} = \text{real}(\_\text{Real} + \text{int}(\_\text{Int}) \text{ when } _{\text{Expr1}} = \_\text{Real}, _{\text{Expr2}} = \_\text{Int}. \]

- Combinations of datatypes in functions which are not meaningful are evaluated to "null". Since there is no default mechanism in the ASF + SDF formalism, we had to write out nearly all erroneous combinations, see for example the equation [add6] - [add25] (we have not completed this for many other combinations).

- The treatment of null in predicates is worked out completely (conform the tri-state truth tables of SQL; see §5.18 <search condition> General Rules 2 with null serving as the notion unknown).

Module Tuples defines the several types of tuples we need; a module parameterized with the objects to put in the tuples would be preferable. Beside tuples, this module also implements the promised function bind(INST_TUP,EXPR) -> EXPR, the first argument being a tuple of variable instances (pairs VAR ":" VAL) and the second an expression to evaluate. As an example we show the following reduction:

Example 2
bind(<i : 4, b : true, s : 'actual'>, (i * i > 16) & b & s = 7) -> ... ->
(4 * 4 > 16) & true & 'actual' = 7 -> ... ->
false & true & null -> ... -> false

Writing down the equations for this binding is, again, just working down all the cases which form an expression. Two other useful functions are implemented:
- inst(VAR_TUP,VAL_TUP) -> INST_TUP (instantiate variables with values) and
- val(EXPR_TUP) -> VAL_TUP (conversion of tuples with expressions to tuples with single values by implicit evaluation of the expressions).

The following serves as an illustration of these last two functions.

Example 3
Suppose we want to extract information from a relation Employee with name and salary as attributes. At some point in this process we reach the employee <Jones, 100>. The information to be derived is the employee's name and salary after an increase of 10%. The following reduction will then occur:

val(bind(inst(<name, salary>, <Jones', 100>), <name, salary * 1.10>)) ->
val(bind(<name: Jones', salary: 100>, <name, salary * 1.10>)) ->
val(<Jones', 100 * 1.10>) ->
<Jones', 110>
Before discussing our specification for relations and more, we would like to sketch how to add a newly defined datatype, e.g. DATE, to the system:

1. specify the datatype of sort DATE in one or more modules with top-module Dates,
2. import Dates into Values and add "DATE -> VAL" in the context-free syntax section,
3. import Dates into Expressions and, for each operator or function, write out all the combinations with at least one argument of sort DATE and determine the corresponding result,
4. add the equation "[...] bind(<Var : _Date, _Insts>, _Var) = _Date" when _Date is a variable of sort DATE, and
5. when a function of sort DATE has to be lifted to a function of sort EXPR, which e.g. could be the case with day-of-week(DATE) -> STR, this function has to be declared and specified in Expressions and the implementation of the function bind() must be expanded too.

3.4 Relations, Databases

Relations, with sort name REL for short, are defined in module Relations. Constants of this sort follow the notation given in §2.1. The traditional set operations are all implemented, except for "/".

The unique function "!" REL -> REL is provided to remove duplicate tuples from a relation.

The function VAL_TUP "@" REL -> BOOL should be read as the traditional 'element of' (a ∈ S) and EXPR_TUP "@" REL -> EXPR is added to the sort EXPR for allowing subqueries to be made (see below). The function "#" REL -> INT gives us the cardinality of a relation.

Two quantified predicates are provided by the functions

\[
\text{for_all \ Var_TUP "@" REL : EXPR -> BOOL and there_is \ Var_TUP "@" REL : EXPR -> BOOL,}
\]

denoting the traditional predicates "∀x ∈ V: p(x)" resp. "∃x ∈ V: p(x)".

Example 4

\[
\text{for_all <x> ∈ \{<3>, <4>, <5>\} : x > 2 \to \ldots \to \text{true}}
\]

The largest element (tuple) of a relation can be extracted using sup(REL) -> EXPR, the smallest using inf(REL) -> EXPR and the summation of all values of a single column, provided they are all integers or reals or strings (as should always be the case since domains may not be mixed), is done by sum(REL) -> EXPR.

A selection from a relation can be made using \{VAR_TUP @ REL | EXPR\} which could best be read as a variant of the more common set definition, for example \{(<x,y> ∈ ℝ × ℝ | x^2 + y^2 = r^2)\}.

This defines a set of coordinates in the two-dimensional plane: a circle with radius r and middlepoint <0, 0>. In our case however the set to draw values from is always finite.

The selection \{<x> @ \{<1>, <2>, <3>\} | n <> 2\} results in a relation \{<1>, <3>\}.

The generalized projection function from the introduction can be made with the function

\{EXPR_TUP ← VAR_TUP @ REL\}

and is best illustrated by giving an example.

Example 5

\{<name, salary, salary * 1.10> ← <name, address, salary, start_date> @ Employee\}

This query will result, provided that a constant relation as intended is inserted in place of Employee, in a listing of the type, old salary and new salary (after a raise of 10%) of all employees.

Combining both functions gives us the function

\{EXPR_TUP ← VAR_TUP @ REL | EXPR\}.
Example 6
\[
\begin{align*}
&\{<\text{name}, \text{salary}, \text{salary} * 1.10>\} \subset\ \\
&\{<\text{name}, \text{address}, \text{salary}, \text{start\_date}>\} \subset\ \\
&\text{Employee} \mid \text{start\_date} < 1:1:91\}
\end{align*}
\]
This will result in the same listing as before, but only of employees who were hired before Jan 1, 1991.

The function \{ VAR\_TUP @ REL \} is defined for those who like to simulate the SQL query "SELECT * FROM R".

The first equation of module Relations, which is commented out for our purpose, can be used as a normaliser: duplicate tuples are found and removed from a constant relation. We saw this before in §3.3 and closer inspection tells us again that the equality of tuples is based on the equality of all corresponding values. This is not valid however when some datatype does not provide one and only one normal form for two different representants of the same equivalence class. This turns out to be the case for Reals: "004.600" is not textually equal to "4.6". Therefore we defined the equality of tuples in module Tuples and we used it in the implementation of VAL\_TUP @ REL -> BOOL.

The implementation of the relational operators is quite straightforward:
1. each time a tuple with values from the relation is taken,
2. the attributes are instantiated with these values,
3. the selection expression (when provided) is evaluated to see whether or not the tuple is selected, and finally, when the tuple is selected,
4. the expressions in the projection tuple (when provided) are evaluated to form the output value tuple.

Example 7 (compare also with the sample database)
\[
\begin{align*}
&\{<\text{sport}, \text{amount} * 1.10> \subset <\text{sport}, \text{amount}> \subset \\
&<\{<\text{Zwemmen}', 50.00>, <\text{Tennis}', 75.00>, <\text{Wandelen}', 1.50>\} \mid \text{sport} \not< \text{Tennis}> \} \rightarrow \\
&\{\text{val}(\text{bind(<sport>: 'Zwemmen', amount: 50.00>, <sport, amount * 1.10>))) \} + * \\
&\{\text{val}(\text{bind(<sport>: 'Wandelen', amount: 1.50>, <sport, amount * 1.10>))) \} \\
&\{\text{val(<Zwemmen', 50.00 >}) + \{\text{val(<Wandelen', 1.50 >}) \} \rightarrow \\
&\{<\text{Zwemmen}', 55.00>, <\text{Wandelen}', 1.65>\} \rightarrow \\
&\{<\text{Zwemmen}', 55.00>, <\text{Wandelen}', 1.65>\}
\end{align*}
\]

*) Since
\[
\text{bind(<sport>: 'Tennis', amount: 75.00>, sport \not< 'Tennis'>) \not< 'Tennis' \not< 'Tennis'> \rightarrow \text{false,}
\]
the tuple \{Tennis', 75.00> will not be selected.

When evaluating a subquery: \{<x> \in X \mid <x> \in \{<y> \in Y \mid x = y >\}\}, resulting in those values of X which occur in Y as well, we need to bind the variables in the inner expression x = y with values from the outside. If for example x has value 4 from the relation X, the bound predicate should look like "<4> \in \{<y> \in \{... | 4 = y >\}". Therefore we extended the equations for bind(INST\_TUP, EXPR) -> EXPR with the case that the argument is such a predicate and introduced another bind function rmbind(INST\_TUP, REL) -> REL, which substitutes variables for values inside a relational expression.

If we had chosen the same variable names in the example above, as in \{<x> \in X \mid <x> \in \{<x> \in Y \mid x > = 0 >\}\}, we surely don’t want to substitute a value for the first x into the expression x >= 0. Function scope-cover(INST\_TUP, VAR\_TUP) -> INST\_TUP removes instantiated variables from the instantiate tuple when they occur in the variable tuple given.
Finally, we reach module RDBS (acronym for Relational DataBase System), which defines the syntax and semantic needed to write down and evaluate a schema followed by an actual database and a program (alltogether an 'EXE'). Note from the equations that we do not need to encapsulate an EXE in an evaluation function. Whenever there is at least one statement in the program, it is evaluated automatically after pressing the reduce button.

The only possible statements in a RDBS program are assignments to a relation in the database, so the only form of output to the end-user is an updated database after execution of the program. Note that there is absolutely no checking whatsoever: the schema is not used for the execution of a statement, an assignment to a non-existing relation is cannot be done, use of an unknown relation is not prohibited, etc.

The main functions of this module are erel(SCHEMA, DBASE, REL) -> REL used to instantiate relation variables with constant relations from the database and eval(SCHEMA, DBASE, EXPR) -> EXPR to do the same, but with relations occurring inside a (value) expression.

Two types of queries are added to the syntax:

{EXPR_TUP <- @ REL \ EXPR} and

{@ REL \ EXPR}.

They look like queries without a variable tuple indicating the attributes of the relation and so they are indeed. This type of query can be used to translate SQL queries; the corresponding attributes are looked up from the schema by using function attr(SCHEMA, REL) -> VAR_TUP. (Note: this is the only case where a schema is actually used.)

3.5 Return to the example

Here we return to the sample of §3.1. The result of the program applied to the database is a changed database and an empty program, as shown below:

```
schema
  Sports  == {? Zwemen\>', '<Tennis\>', '<Wandelen\>', '<Voetbal\>'
  Numbers == {? n@ Integers | (n >= 800000) & (n <= 919999)
  Amounts == {? x@ Reals | (amount >= 0.0) & (amount <= 300.0)
  Students == {? number, name, phone@ Numbers # Strings # Strings
  Members == {? number, sport@ Numbers # Sports
  Contributions == {? sport, amount@ Sports # Amounts
  Account == {? name, phone, amount@ Strings # Strings # Amounts
  Swimmers == {? name@ <number, name, phone@ Students

database
  Students = {
    <891247, 'Al-Hassan', '02990 - 42185'>,
    <893218, 'Bakker', '020 - 6123316'>,
    <896673, 'Jansen', '020 - 6123316'>,
    <910001, 'Nieuwland', '015 - 147203'>
  }
  Members = {
    <891247, 'Tennis'>,
    <893218, 'Zwemen'>,
    <893218, 'Tennis'>,
    <910001, 'Zwemen'>
  }
  Contributions = {
    '<Zwemen\>', 55.00 >,
    '<Tennis\>', 82.50 >,
    '<Wandelen\>', 1.65 >
  }
  Swimmers = {?<name@ <n1, name, p, n2, sport@ Students # Members \ (n1 = n2) & (sport = 'Zwemen')

  Account = {
    <891247, 'Al-Hassan', 82.50>
    <893218, 'Bakker', 55.00>
```
In this output database we see the changes to Students, Members and Contributions:

- addition of student 'Nieuwland' to Students,
- deletion of student 'The' from Students,
- enrollment of student 910001 ('Nieuwland') with sport 'Zwemmen' to Members, and
- increase of contributions (amount in relation Contributions) with 10%.

The relation Account is the list of payments to be done after the increase of contributions. (Note that in SQL it would have been possible to sum the amounts to be paid by student Bakker by using a grouping on student name or -number; this is not possible using relational algebra only.)

Finally, we see the effect of using a view by inspecting the contents of relations X and Y (Swimmers before and after enrollment of student 'Nieuwland' for 'Zwemmen'). In this case the administration would have been very pleased if the database designer had defined Account as a view also.

With just a view additions to this database system, like dynamic creation and destruction of relations, it has all the power to implement a ready-to-use relational database system in practice.
4 Syntax of SQL

In this chapter we describe the SQL syntax based on the standard definition. First, the syntax from the BNF definitions is translated to a corresponding SDF syntax definition. Next, some preliminaries about typechecking SQL are given. Finally, typecheck functions for SQL schema's and query specifications are explained.

4.1 Conversion of BNF to SDF

The first step in specifying SQL is to translate the BNF definition given in the standard into lexical and context-free syntax of SDF. A main choice has to be made: whether to stay close to the BNF definition or to rearrange this syntax completely. Because the typecheck rules given by the Syntax Rules sections are very much related to the BNF definition, we have chosen for the first option.

The following steps describe how we converted the BNF definition, listed in Appendix A, into SDF.

1. A separation had to be made which language constructs to put in the lexical syntax part and which in the context-free syntax part. The main point of distinction here is whether or not layout is allowed between (non) terminals of the definition. Combining the BNF definitions and Syntax Rules of the standard, we decided to place \(\S 5.1 - \S 5.3\) (\(<\text{character}\), \(<\text{literal}\>\ > \ <\text{token}\>) in the lexical syntax part.

2. All non-terminals of the BNF definition have been rewritten to form practical SDF sortnames: supply words with leading capitals, put words together (i.e. removing non-letters), and finally remove the surrounding brackets (so \(<\text{table name}\>) is converted to TableName). When these sortnames became too long, we abbreviated them, preferably in the common way (e.g. Identifier was shortened to Ident).

3. All terminals had to be surrounded by double quotes (e.g. \(<\text{mantissa}\>E<\text{exponent}\>) is translated to Mantissa "E" Exponent).

4. Since ASF+SDF does not allow context-free functions to contain alternatives, optionals and grouping we had to rewrite these constructs, and repetitions as well, in the following way:

\[
\begin{align*}
\text{NonTerm} & ::= T_1 | T_2 | \ldots | T_n & \rightarrow & \text{NonTerm} ::= T_1 \\
& \quad \quad \quad \quad \quad \quad & \rightarrow & \text{NonTerm} ::= T_2 \\
& \quad \quad \quad \quad \quad \quad & \rightarrow & \text{NonTerm} ::= T_n \\
& \quad \quad \quad \quad \quad \quad & \rightarrow & \text{NonTerm} ::= T^* \\
& \quad \quad \quad \quad \quad \quad & \rightarrow & \text{NonTerm} ::= e_1 e_2 e_3 \\
& \quad \quad \quad \quad \quad \quad & \rightarrow & \text{NonTerm} ::= e_1 e_3 \\
\end{align*}
\]

5. Grouping in extended BNF definitions, surrounded by braces \(\{\}\), has in general to be replaced by a subsort. But in this situation it was not necessary to do so, because of the following reasons:

- the syntax occurring in groups could be specified using lexical syntax only,
- it were lists of simple alternatives (and were written out as above), and
- the combination \(<\text{non terminal}\> \{\ldots, \text{<non terminal}> \ldots\}\) was replaced by the SDF comma-list construction \([\text{NonTerm},\ldots]\).

6. Rules like \(\text{NonTerm} ::= E\) were reversed to \(E \rightarrow \text{NonTerm}\)

The sections of the definition we decided to become lexical syntax was treated different from the rest:

- the definitions for \(<\text{character}\>, \ <\text{digit}\>, \ <\text{letter}\>, etc are incorporated in the other definitions (e.g. by replacing \(<\text{digit}\>\) with \([0-9]\)),
- a string literal does not follow the rules of the definition exactly because the \(<\text{quote representation}\>\ > \ <\text{character string literal}\>) met some difficulties (note that the empty string is not part of the SQL language, one should use NULL instead),
- \(<\text{comment}\>, \ <\text{space}\>\ > \ <\text{newline}\>) are translated to define the predefined sort LAYOUT, and

- 29 -
• the tokens and keywords of the language are derived from the SDF syntax automatically and in such a way that it follows the rules of the definition (the SDF 'prefer literals' rule for instance); therefore the sorts <token> and <keyword> are not specified in SDF.

7. In order to form a proper SDF module, we had to add a sorts section (by listing all distinct output sortnames of the syntax functions) and a variables section for later use (with names derived from the sortnames).

The final stage of translating the BNF definition of SQL is to split up the syntax so far into handy-to-use smaller modules. Figure 2 shows the import graph of these modules:

![Diagram of the import graph of SQL modules](image)

Figure 2: Import graph SQL language.

Lex and Names: the lexical syntax from §5.1 - §5.3 and the 'names' from §5.4 (because these names occur practically everywhere).

ValueExpressions: all language constructs (§5.6 - §5.9) to define expressions resulting in a single value of some datatype (as opposed to TableExpressions and Predicates).

Predicates: the expressions resulting in a truth value true, false, or unknown (§5.10 - §5.17).

QuerySpecs: the main query function of SQL (§5.18 - §5.25).

DataTypes: the character string and various numeric datatypes of SQL (§5.5).

Schemas: the schema definition language to define schemas, tables, views, and privileges (§6.1 - §6.10).


Modules: module language of SQL (cursor and procedure declarations) (§7.1 - §7.3).

Note: DataTypes is the only module which contains equations; they implement automatically the Syntax Rules 1 and 3 of §5.5 <data type>.
The import of module QuerySpecs in module Predicates was also needed (dotted arrow in Fig. 2), since predicates can contain subqueries. A cyclic import is not allowed in ASF + SDF, so we left out this import. To avoid the error message 'sort SubQuery not defined' when dealing with predicates, we declared the sortname SubQuery in module Predicates and delayed the definition of this sort until module QuerySpecs. So, however it is possible actually, neither of these modules should be used without the other!

4.2 Typechecking preliminaries

Typechecking is the process of verifying fragments of a program source text with respect to information from an environment. In general we need a function: tc: ENV # PROG -> BOOL. The environment is usually a quickly accessible description of previously declared symbols (such as a symbol table containing declared identifiers with their type). During the process of typechecking we break down the language constructs into smaller pieces and check these smaller pieces accordingly. So we end up with a collection of functions, each capable of checking some construct in the language. We can either give them different names, like tc_prog, tc_stat etc, or use the overloading mechanism and call them all tc.

During this top-down typechecking process, we sometimes need to give extra information down to lower levels (so-called inherited attributes) and sometimes we gather information at lower levels (synthesized attributes) and pass this as a result back to the caller. Due to these synthesized attributes some typecheck functions need more than one output sort, like in tc: ENV # EXPR -> BOOL # TYPE.

A proper typecheck function shall not result in just "false" when the source text is found incorrect. Instead some sort of message about the kind of error detected and the location where the error took place is convenient for the user. (Beside pinpointing where the error took place, the ideal typecheck function tries to find out the reasons for an error and suggests possible solutions to fix the problem.) More than one of such a message must be given if more errors are detected in the source. We must however avoid winding up in generating errors due to errors previously found.

In the next paragraphs we shall describe a typecheck function for SQL. Not all of the language constructs of SQL have been done; only a significant subset. The constructs we have done are §5.1 - §5.25 (i.e. a <query specification> and all its components) and §6.1 - §6.6 (i.e. a complete <schema>) with exclusion of the following:

- column constraints, default clauses, table constraints, and privileges are not checked,
- parameters (SQL modules), and variables (embedded SQL statements) are excluded,
- argument and result of set functions (COUNT, SUM, etc.) are checked only for type,
- grouping of tables (select list group-by-, and having-clause) are checked only limited.

Facing the problem of typechecking SQL we have made the following choices:

1. Choosing the environment datatype:

As environment-parameter for a typecheck function for query specifications we use a more easily manageable and quicker accessible datatype than the language construct schema itself (or more accurate: the self chosen environment). An environment can be translated by the function envdescr(Environrment, QuerySpec) -> EnvDescr from module SchemaDescr. Next, such a resulting environment description can be checked against the syntax rules from §6.1 - §6.6 by using the function envcheck(EnvDescr) -> ERROR. An OK-checked environment description can now be used for typechecking a series of queries typed by a user, as is done in module TypeCheck.

Using an environment description instead of an environment itself is easier manageable because of some reordering of entries (like privilege definitions separated from table- and view definitions) and because there is more uniformity (see for instance a description of a datatype: DTDescr). It is quicker accessible because there are less cases to be dealt with.
This choice however has one major drawback. When we want to convince someone that our typechecking function is conform the definition in the standard, we need to demonstrate that all syntax-rules are correctly checked based on a description and not on the source text itself. (Note that the standard defines descriptions of some language clauses.)

SDF allows us to incorporate language clauses such as Identifiers, Statements, or whatever, inside descriptions, simply by inserting them in the definition of the structure. We considered this new opportunity as a very practical feature.

2. Choosing the error message mechanism:

Instead of a fancy yet complex error message mechanism, which is beyond the scope of this thesis, we have chosen to present just simple and adequate error messages. Not just error messages in the form of strings, but we invented the type ERROR (see module Errors) in the following way:

• OK is the ERROR constant denoting that all is well.

• Mixing explanatory text with relevant language clauses is possible, as in:
  "Column" ColumnSpec "undefined!" -> ERROR
  in an equation:
  [...] check(_Env, _ColumnSpec) = Column _ColumnSpec undefined! when ...
  Additional information can be added to the message by providing more context:
  [...] check(_Env, _ValueExpr, _ColumnSpec) =
  Column _ColumnSpec undefined in _ValueExpr! when ...

• We used the binary operator ERROR "AND" ERROR -> ERROR to combine errors of mutual independent checks, as when checking lists of value expressions in a select list.

• The binary operator ERROR "AND" "THEN" ERROR -> ERROR is used to combine checks which are dependent in time, for example the query
  SELECT * FROM R WHERE R.A <> 0
  will result in an error message:
  Table Ref R undefined!
  instead of:
  Table Ref R undefined! AND Column R.A undefined!
  when R is not a table from the environment.

3. Choosing the moment to translate:

While typechecking language constructs, it is very tempting to look at the translation of these constructs as well. In many situations it is even a shame not to use the information determined during the typechecking. For instance when encountering an expression 'IntVar + RealVar', the typecheck function knows that a conversion of the value of the integer variable IntVar to a real is needed. This information can be passed as a synthesized attribute for use by a translator. A translator based on the same unmodified language construct, has to determine again the type of both variables and the conversions needed.

We choose, however, not to think of translation during typechecking for the following reasons:

• Keep the size of the specification relatively small. Having the syntax and equations of both languages read into the system, and all typechecking and translation functions as well, it would soon be too large to execute.

• Since the target language of the translation of SQL is very flexible (the above case is dealt with automatically), we don't need so much 'type' information while translating.

• A clear separation between typechecking and translation is desirable when we want to anticipate on the possibility to translate SQL to other relational database languages than RDBS.
4. How to deal with views:

A view definition contains a query specification and can thus refer to other tables (or views) in its very definition, for example:

```
CREATE VIEW SWIMMERS (NAME) AS
    SELECT NAME FROM STUDENTS, MEMBERS
    WHERE (STUDENTS.NUMBER = MEMBERS.NUMBER) AND
    (MEMBERS.SPORT = 'Zwemmen').
```

This leads to the following three consequences:

- We have to use all the syntax and semantics of checking query specifications for checking a schema too, which is not a problem when all functionality is available already, but has been somewhat difficult to design.

- When a column name in an expression is encountered during typechecking, one of the things to find out is to which table or view this column belongs and of what type it is. If the column belongs to a view, we have to dig out the type information from the query defining the view. For this reason we need to convert a view definition to something resembling an ordinary (base) table definition (see also §6.9 Syntax Rule 7 requiring such a description). Since a view definition can refer to other tables and views, we can only do this after checking the whole environment first.

- A view definition can refer directly or indirectly to itself, e.g. two views defined as queries upon each other; we did not try to solve this possibility (there are also no corresponding Syntax Rules in the standard).

The total typecheck function we implemented for SQL Schema's and QuerySpecs has been divided over several modules, as shown in Figure 3:

![Diagram of TypeCheck facility](image)

**Figure 3:** Importgraph of TypeCheck facility.
4.3 Typechecking SQL schema's

We will now describe the typechecking modules for SQL Schema's, SchemaDescr and SchemaCheck.

**SchemaDescr**
- definition of the sort EnvDescr, the datatype for a description of an environment, and all its subtypes, and variables of sort EnvDescr,
- specification of the function env-descr(Environment) \(\rightarrow\) EnvDescr (and functions for describing lower language constructs in the hidden section),
- two auxiliary functions lookup-table() and lookup-column().

An important datastructure is a so-called TVListDescr, which is a list of descriptions of table- or view definitions. This list is not only constructed when describing a schema and used for lookup of a table name, but also for determining the context of a value expression, i.e. what tables can be referred to using column names.

The result of function lookup-table(TableName, EnvDescr) \(\rightarrow\) TVListDescr is defined as all those tables whose name satisfies the TableName. For example table name R satisfies both S1.R and S2.R for schema identifiers S1 and S2; see also equations [eqn1] - [eqn4]. So given a table name, lookup() can find zero or more corresponding tables and thus we defined the output sort as a TVListDescr. The same idea holds for lookup-column(ColumnSpec, TVListDescr) \(\rightarrow\) ColListDescr, which finds zero or more columns given a column name and a list of tables.

Another heavily used structure is a DTDescr, the description of an SQL datatype. This is a tuple containing type (String, Exact, Approx or Notype), length or precision (Unsigned), and optional scale (Unsigned), e.g. \langle\text{approx, }5, 2\rangle\. When no particular datatype can be determined, usually the 'constant' \langle\text{Notype, }0, 0\rangle is returned.

To determine a datatype we use the functions type(), length() and scale(). Type() determines the type description belonging to a datatype by simply matching with datatype's subsorts (equations [type1] - [type3]). Length() determines the length of a string type or the precision of a numeric type and scale() the optional scale of a numeric type.

The latter two functions do not need to specify all the cases a datatype can look like, because the equations of module DataTypes take care of abbreviations and defaults. So a possible reduction path is:

\[
\text{length(CHAR)} \rightarrow \text{length(CHARACTER)} \rightarrow \text{length(CHARACTER}(1)) \rightarrow 1.
\]

**Example 1**

```
env-descr (  
  CREATE ENVIRONMENT UNIVERSITY  
  CREATE SCHEMA ADMINISTRATION  
  CREATE TABLE SPORTS (  
    SPORT CHAR(10)  
  )  
  CREATE TABLE STUDENTS (  
    NUMBER NUMERIC(6, 0)  
    NAME CHAR(20)  
    PHONE CHAR(15)  
  )  
  CREATE TABLE MEMBERS (  
    NUMBER NUMERIC(6, 0)  
    SPORT CHAR(10)  
  )  
  CREATE TABLE CONTRIBUTIONS (  
    SPORT CHAR(10)  
    AMOUNT NUMERIC(5, 2)  
  )  
  CREATE VIEW SWIMMERS (NAME) AS
```
SELECT NAME
FROM STUDENTS, MEMBERS
WHERE STUDENTS.NUMBER = MEMBERS.NUMBER
AND SPORT = 'Zwemmen'

-> < UNIVERSITY,
    [ < ADMINISTRATION,
       [ < ADMINISTRATION, SPORTS,
          [ < SPORT, String, 10, 0>, NULL, false> ],
          []
       >
       < ADMINISTRATION, STUDENTS,
          [ < NUMBER, Exact, 6, 0>, NULL, false>,
            < NAME, String, 20, 0>, NULL, false>,
            < PHONE, String, 15, 0>, NULL, false> ],
          []
       >
       < ADMINISTRATION, MEMBERS,
          [ < NUMBER, Exact, 6, 0>, NULL, false>,
            < SPORT, String, 10, 0>, NULL, false> ],
          []
       >
       < ADMINISTRATION, CONTRIBUTIONS,
          [ < SPORT, String, 10, 0>, NULL, false>,
            < AMOUNT, Exact, 5, 2>, NULL, false> ],
          []
       >
       < ADMINISTRATION, SWIMMERS,
          [ NAME ],
          SELECT NAME
          FROM STUDENTS, MEMBERS
          WHERE STUDENTS.NUMBER = MEMBERS.NUMBER
          AND SPORT = 'Zwemmen',
          false
        ],
        []
      >
    ]
  >

X

SchemaCheck
- specification of the function tc(EnvDescr) -> ERROR, which checks a description of an
  environment and resulting in either OK or some applicable error message(s),
- specification of expand-env(EnvDescr, EnvDescr) -> EnvDescr, which converts a description of a
  view to a description of a (base) table after the typecheck has succeeded.

Note the arguments to expand-env(): the first EnvDescr is used as a lookup for expansion of the
second one, resulting in the output EnvDescr. So when calling this function, it is preferred to use the
same argument twice.

Example 2
tc(env-descr(
    CREATE SCHEMA S
    CREATE TABLE A.T (
C CHAR(1)
C NUMERIC(1)
)
CREATE TABLE S.T ( 
C CHAR(1) 
)
))

Possible errors are:
• Authorization Id A should be S!
• Table Name T should be unique!
• Column Name C should be unique! (only in first table)
Only the first of these errors will be returned in our implementation.

Example 3
In a number of steps we show the typechecking of the view Swimmers from the sample database.

tc(env-descr ( 
... 
CREATE VIEW ADMINISTRATION.SWIMMERS (NAME) AS 
SELECT NAME 
FROM STUDENTS, MEMBERS 
WHERE STUDENTS.NUMBER = MEMBERS.NUMBER 
AND SPORT = 'Zwemmen' 
... 
)
The error messages
• Authorization Id ADMINISTRATION should be ADMINISTRATION!
and
• Table Name SWIMMERS should be unique!
will not occur, since the corresponding conditions of equations [tv2] and [tv3] fail.

At this point equation [tv3] will come into action: query-check(E, Q), where E is the environment
description and Q the query specification from the view definition, will result in a list column
descriptions (one in this case):
[<NAME, <String, 20, 0>, NULL, [], false>],
provided that Q is OK-checked (see below for query-check()).

This list column descriptions is matched against the view column list, "(NAME)", using function
coll-check(). The check to be performed:
should-provide-viewcolumnlist() AND THEN
all-ids-must-be-different() AND THEN
right-number-of-viewcolumns()
will yield OK, since
• a view column list is provided,
• NAME is the only identifier in this list, and
• the query specification Q results in a single column.

After expansion using expand-env() this view definition will have as description:
<ADMINISTRATION, SWIMMERS, [NAME, [NAME, <String, 20, 0>, NULL, [], false>], []>. 

- 36 -
4.4 Typechecking SQL query specifications

In this section we explain the modules for typechecking SQL queries.

TC_ValueExpr
- definition of sort VE_EDP (ValueExpression_ErrorDescriptionPair) and
- function ve-check(TVListDescr, ValueExpr) -> VE_EDP.

The function ve-check() checks a given value expression against the rules from the definition and with respect to the context TVListDescr. The latter argument is mostly derived from a list of tables in a from clause as part of a query specification and mainly used for looking up a given column in this list. Note the use of lookup-column() to see whether a column specification refers to a unique column, an ambiguous column, or no column at all in the given context (equations [c1] - [c3]).

The result of ve-check() applied to an expression is either an error message not equal to OK and some undefined datatype, or an OK message and the description of a datatype.

Note: it turned out to be very difficult to determine the precision and scale of an exact numeric literal. The precision is defined as the number of digits and the scale as the number of digits right from the (implicit) decimal point ($\S 5.2$ <literal> Syntax Rule 4). Our implementation is only partial correct.

Example 4

SELECT STUDS.NUMBER FROM STUDENTS, MEMBERS

The select list STUDS.NUMBER of this query about our sample database, can result in the following typecheck error messages:
- Type of operand(s) of '+' may not be string!
- Column STUDS.NUMBER undefined!
- Column NUMBER ambiguous! (after removal of "STUDS.")

Example 5

Suppose someone is interested in the following query about our sample database:

SELECT AVG(AMOUNT) + 1.0 FROM CONTRIBUTIONS.

With the current values in our database, this query will result in the value 43.17.

First, we show how the exact literal 1.0 will be checked:

ve-check(1.0) -> [ve11]
<OK, <Exact, precision(1.0), scale(1.0)>> -> [prec3] & [scale1]
<OK, <Exact, 2, 1>>

Also, the column reference AMOUNT will be checked:

ve-check(AMOUNT) -> [ve13]
<Exact, 5, 2>>
<OK, <Exact, 5, 2>>

(*) Since lookup-column(AMOUNT, ...) results in exactly 1 column description, AMOUNT is defined and unambiguous.

Since the error part of the last result is OK and the type is not String, AVG(AMOUNT) will be checked using equation [ve26]:

ve-check(AVG(AMOUNT)) -> [ve26]
<OK, <Exact, 5, 2>>

The value expression AVG(AMOUNT) + 1.0 will be checked:

ve-check(AVG(AMOUNT) + 1.0) -> [ve1]
exp-check('+', ve-check(AVG(AMOUNT)), ve-check(1.0)) -> (see above)
exp-check('+', <OK, <Exact, 5, 2>>, <OK, <Exact, 2, 1>>> -> [exp4] (*)
<OK, <exp-type(Exact, Exact), exp-prec(5,2), exp-scale('+', 2, 1)>> -> [etyp1],[epr1] & [esc1]
<OK, <Exact, 5, 2>>
(*) Since both errors are OK and none of them are of type String.

So the value expression given in the query is type correct (OK) with type exact numeric, precision 5 and scale 2; therefore the result is 43.17 and not 43.16666 for instance.

Just like the distribution of the SQL language over several modules, we would like to divide the typecheck function into modules TC_Predicate and TC_Query. For typechecking subqueries in predicates, we need the functionality to check queries and vice versa. In order to solve this cyclic use, we created the modular hierarchy outlined in Figure 3 (dotted lines). Module TC_QueryPred_syn defines all the common syntax for typechecking both queries and subqueries. Modules TC_QuerySem and TC_Pred_sem implement the semantics (equations) of this syntax (and local functionality as well). Module TC_QueryPred finally combines these modules in order to form one coherent signature to the outside world.

**TC_QueryPred_syn**
- definition of several output tuples to hold an error and synthesized attributes
- syntax of several check functions, one for each main language clause

**TC_Pred_sem**
The main function of this module is pred-check(EnvDescr, TVListDescr, Predicate) -> ERROR, which just handles all the cases a predicate can look like. The second argument, TVListDescr, stems from a preceding from clause: it determines the context which table- and column names can be used. The first argument, EnvDescr is not used, but handled over to a possible subquery in the predicate (see for instance an InPredicate: ValueExpr IN SubQuery).

**Example 6**
```
SELECT *
FROM MEMBERS
WHERE N NOT IN (SELECT NUMBER, NAME FROM STUDS)
```

This query about our sample database, finding non-students who are enlisted for some sport (e.g. staff or deregistered students), can result in the following messages. A possible correction for the errors is also given:
- Table Ref STUDS undefined! (change STUDS to STUDENTS)
- SubQuery must result in a single column! (remove "NUMBER, ")
- Types are not comparable! (change NAME to NUMBER)
- OK

A useful function for checking a comparison predicate like "A + 4 > 5.0" is comparable(VE_EDP, VE_EDP) -> ERROR.

Given two results from typechecking value expressions, i.e. two tuples <ERROR, DTDescr>, this function determines as output (see equations [comp1] - [comp8]):
- an error message when either one or both operands are incorrect, or
- an error "Types are not comparable!", when this is the case, or
- "OK" when both operand are correct and comparable.

The message OK is returned when both operands are correctly typed and the function comparable(DTDescr, DTDescr) -> BOOL returns true. Note that the latter reduces to "true" for all the cases for which both datatypes are comparable. For all (11) other cases no equation is given, so they are in normal form already. Therefore we need to test for "comparable() != true" instead of "comparable() = false".

**Examples 7** (omitting parameters _EnvDescr and _TVListDescr)
```
pred-check(SPORT = 'Zwemmen') -> [pred1]
```

- 38 -
comparable(ve-check(SPORT), ve-check('Zwemmen')) -> (see TC_ValueExpr)
comparable(<OK, <String, 10, 0>>, <OK, <String, 7, 0>>) -> [comp3]
OK
The last reduction because the condition succeeds:
comparable(<String, 10, 0>, <String, 7, 0>) = true -> [comp4]
true = true.

pred-check(SPORT = 4)
would result in (see [comp2]):
Types are not comparable!
since there are no equations applicable for comparable(<String, 10, 0>, <Exact, 1, 0>).

pred-check(NEAR = FAR)
would result in (see [comp1]):
Column NEAR undefined! AND Column FAR undefined!
since the check of both value expressions (column references) will return non-OK errors.

The typecheck of a like predicate (equation [pred10]) is a nice example of the use of "AND" and "AND THEN" for errors.

Example 8 (again omitting some parameters)
pred-check(NAME LIKE 4 ESCAPE \) -> [pred10]
must-be-string(NAME) AND must-be-string(4) AND
(must-be-string(\) AND THEN must-be-length(\)) ->
OK AND Type of 4 must be a string! AND
(OK AND THEN Value \ must be a single char!) ->
Type of 4 must be a string! AND Value \ must be a single char!

TC_Query_sem
Checking a query specification:

SELECT SelectList
FROM {TableRef "","}+
WHERE SearchCondition
GROUP BY {ColumnSpec "","}+
HAVING SearchCondition

is the task of function query-check(EnvDescr, QuerySpec) -> QUERY_EDP and subsequent 'deeper'
functions.

In very general terms the checking process is as follows:

1. texpr-check(EnvDescr, TVListDescr, TableExpr) -> TEXPR_EDP
   Checks the table expression (starting at keyword FROM in the query) in the way described below.
   The second argument of this function is a list of tables, possibly from a preceding subquery and is
   used to determine if a column specification is an outer reference or not (as defined in §5.7 <column
   specification> Syntax Rules 4). For a start query-check() will provide an empty table list to this
   function.

a. from-check(EnvDescr, FromClause) -> FROM_EDP
   using table-check(EnvDescr, TableReference) -> TABLE_EDP
   Checks all tables mentioned in the from clause (FROM {TableRef "","}+) for their unambiguous
   presence in the environment, possibly renamed by a correlation name via correname() from module
   SchemaDescr.
Due to §5.20 <from clause> Syntax Rules 1, a correlation name will always replace the original table name. Therefore "SELECT * FROM R R1 WHERE R.C <> 0" will result in an error since R.C is an undefined column name. When all table references are considered OK, from-check() returns a description list of these tables.

b. where-check(EnvDescr, TVListDescr, WhereClause) -> WHERE_EDP
   using search-check(EnvDescr, TVListDescr, SearchCondition) -> SEARCH_EDP
   Checks the where clause (WHERE SearchCondition), i.e. all the containing predicates. Since column references in these predicates may refer to tables of the immediate preceding from clause, or to tables before that, a combination of the table lists is used for lookup; see function combine-tables() and Figure 4 below.

c. group-check(TVListDescr, GroupByClause) -> GROUP_EDP
   Checks for correct use of containing column references in the group-by clause (GROUP BY {ColumnSpec "",""}+) using the table list from the preceding from clause (and not the combined table list).

d. having-check(EnvDescr, TVListDescr, HavingClause) -> HAVING_EDP
   Checks the containing search condition in the having clause (HAVING SearchCondition), using the combined table list.

The result of from-check() is a combined error from the typecheck functions mentioned in a. - d. and the table list from the containing from clause.

2. sell-check(EnvDescr, TVListDescr, SelectList) -> SELL_EDP
   Checks the select list ("*" or one or more value expressions) in the context of the previously checked table expression (the table list resulting from that function). If a list of value expressions is specified each in turn is typechecked using ve-check() and, when no error is detected, the result is a list of column descriptions (named with special column name ",", not available to users). If "*" is specified a list of column descriptions is assembled of all column names and types from the tables given in the from clause. This list of column descriptions is used to expand the query specifications in the definition of a view and to determine the cardinality and type of a subquery (see functions subval-check() and column-check() of module TC_Pred_sem).

The passing of various table lists, and so implementing the scope rules (see also §5.20 <from clause> Syntax Rules 1 - 4), is shown in Figure 4:

![Diagram](image-url)

Figure 4: Passing table lists while checking a QuerySpec.

Example 9
Define $E$ as the description of the environment of the sample database (see also Example 1 of this chapter) and $S$ and $M$ as the description of the tables STUDENTS resp. MEMBERS.

Further, define $Q$ as the following query:
SELECT NAME FROM STUDENTS WHERE EXISTS
(SELECT NUMBER FROM MEMBERS WHERE
STUDENTS.NUMBER = MEMBERS.NUMBER AND
MEMBERS.SPORT = 'Zwemmen')

We will demonstrate the typechecking of this query as follows:

query-check(E, Q) -> [query3]
<OK AND THEN OK, X, [<NAME, <String, 20, 0>, NULL, [], false>]>  

So Q can be considered type-correct. When necessary, we can extract type information about the result of this query using the column descriptions (e.g. number, name and type of the column).

To show the rewriting of this typecheck function, we need quite a lot intermediate results:

1) from-check(E, FROM MEMBERS) -> [from1] via [table1] and using lookup-table()
<OK, [M]>

2) combine-tables([M], [S]) -> [comb1]
[S M]

3) search-check(E, [S M], STUDENTS.NUMBER = MEMBERS.NUMBER) -> [surch4]
pred-check(E, [S M], STUDENTS.NUMBER = MEMBERS.NUMBER) -> [pred1]
comparable(ve-check([S M], STUDENTS.NUMBER), ve-check(MEMBERS.NUMBER)) -> OK

4) search-check(E, [S M], MEMBERS.SPORT = 'Zwemmen') -> [surch4] & [pred1]
comparable(ve-check([S M], MEMBERS.SPORT), ve-check('Zwemmen')) -> OK

5) using 3) & 4)
search-check(E, [S M], BoolTerm AND BoolFactor) -> [surch2]
<OK AND OK. ->
<OK, >

6) using 5)
where-check(E, [S M], WHERE BoolTerm AND BoolFactor) -> [where1]
<OK, >

7) using 1) and 6)
texpr-check(E, [S], FROM MEMBERS WHERE SearchCond) -> [texpr4]
<OK AND THEN OK, [M]> ->
<OK, [M]>
(Note that [S M] can serve as context for the search condition, but only [M] is passed as a result to form the context for the preceding select list.)

8) using 7)
sell-check(E, [M], NUMBER) -> [sell1]
<OK, [<NUMBER, <Exact, 6, 0>, NULL, [], false>]>  
(Since 7) gave us [M] as context for the select list, NUMBER is an unambiguous reference to a column of MEMBERS.)

9) using 7) and 8)
subq-check(E, [S], (SELECT NUMBER FROM MEMBERS WHERE SearchCond)) -> subq3
<OK AND THEN OK, X, [<NUMBER, <Exact, 6, 0>, NULL, [], false>]>
(The last item in the result tuple is a list column descriptions, telling us that the result of the subquery is a table consisting of a single column. This sole column has characteristics (e.g. name and type) as can be seen in the column description.)
10) using 9)\n\[\text{search-check}(E, [S], \text{EXISTS SubQuery}) \rightarrow [\text{surch4}]\n\text{<pred-check}(E, [S], \text{EXISTS SubQuery}, >) \rightarrow [\text{pred17}]\n\text{<OK, >}\n\]

11) using 10)\n\[\text{where-check}(E, [S], \text{WHERE EXISTS SubQuery}) \rightarrow [\text{where1}]\n\text{<OK, >}\n\]

12) from-check(E, FROM STUDENTS) \rightarrow [\text{from1}] via [table1] and using lookup-table()\n\text{<OK, [S]>}\n
13) combine-tables([S], []) \rightarrow [\text{comb1-2}]\n[S]\n
14) using 11), 12), and 13)\n\[\text{texpr-check}(E, [], \text{FROM STUDENTS WHERE EXISTS SubQuery}) \rightarrow [\text{texpr4}]\n\text{<OK AND THEN OK, [S]>}\n\]

15) sell-check(E, [S], NAME) \rightarrow [\text{sell1}]\n\text{<OK, [NAME, <\text{String, 20, 0}, \text{NULL, [0, false]>}>]}\n(We use the fact that ve-check([S], NAME) \rightarrow <\text{OK, <string, 20, 0>}>)

16) using 14) and 15)\n\[\text{query-check}(E, Q) \rightarrow [\text{query3}]\n\text{<OK AND THEN OK, X, [NAME, String, 20, 0, NULL, [0, false]>}>\n\]

\[X\]

Note: In the typecheck process above, all checks are performed in the conditions of the relevant equations, before the total error message is determined at the right hand side of the equation. This is rather inefficient when some error is detected at an early stage, since in many situations no more typechecking is relevant and thrown away anyway by the AND THEN construct. An alternative is to check the conditions sooner and then to decide to continue typechecking or not.

**TC_QueryPred**
- the main typecheck function tc(EnvDescr, QuerySpec) \rightarrow ERROR

This is simply a projection of the result of query-check() for use as top-level typechecking, for instance for checking a view definition or a query given by an end-user. Once an environment is converted to an EnvDescr (and checked OK), subsequent query specifications typed by an end-user can be typechecked using this function.

**Example 10**
Consider E and Q defined in the previous example, then\n\[tc(E, Q) \rightarrow [\text{tc1}]\n\text{OK}\n\]

**Typecheck**
The total typecheck function of query specifications: tc(Environment, QuerySpec) \rightarrow ERROR, which converts and checks an environment and checks the query specification accordingly.
5 Semantics of SQL

In this chapter we illustrate the meaning of SQL schema's and query specifications. For this purpose we have implemented translation functions from SQL to RDBS and so we are able to express SQL clauses in terms of relational databases and the relational algebra. Also, once we have a translated SQL schema and a proper database, we can execute SQL queries about these databases (i.e. reduce translated queries in module RDBS).

Note: since not all of the functionality of the SQL language has equivalents in the RDBS language, only a limited subset of SQL can be translated.

The translation functions are divided over three modules TR_Schema, TR_ValueExpr, and TR_QueryPred with import relations as shown in Figure 5.

![Diagram of translation modules](image)

Figure 5: Import graph of translation modules.

We will discuss each of the translation modules in turn:

**TR_Schema**
Main function is the translation of an SQL Schema to an RDBS SCHEMA using function tr(Schema) -> SCHEMA. This translation is quite straightforward: each table- and view definition in the schema is converted to a relation definition (RDEF) with attributes and corresponding domains.

**Example 1**

```sql
tr(CREATE TABLE STUDENTS (
    NUMBER NUMERIC(6,0)
    NAME CHAR(20)
    PHONE CHAR(15)
))
```

```sql
schema
STUDENTS <= {<NUMBER,NAME,PHONE> @ Integers#Strings#Strings}
```

**Example 2**

```sql
tr(CREATE VIEW SWIMMERS AS
    SELECT NAME FROM STUDENTS, MEMBERS
    WHERE STUDENTS.NUMBER = MEMBERS.NUMBER
    AND MEMBERS.SPORT = 'Zwemmen'
)
```

```sql
schema
```
SWIMMERS <= {<NAME> @ Strings}
(Note: the view definition must also be converted to an expression in the database part of an RDBS
schema/database/program triplet.)

Note: of course it would have been better if we had specified a translation function for an
environment description.

TR_ValueExpr:
The translation of a value expression in SQL to an expression in RDBS is performed by the function
tr(ValueExpr) -> EXPR. Again, only a limited subset of SQL's value expressions have a counterpart
in RDBS. For instance variables, parameters, set functions, and grouping cannot be dealt with.

Example 3
  tr(Amount * 1.10) -> [ve3]
  tr(Amount) * tr(1.10) -> [ve4] & [ve11]
  Amount * 1.10

TR_QueryPred
Function tr(QuerySpec) -> REL converts an SQL query, and all language clauses that can occur in a
query (including predicates), to a relational expression in RDBS.
Predicates are translated using function tr(Predicate) -> EXPR.

Examples 4 (see also the examples of predicates in §2.2):
  tr(Number = 901274) -> [pred1]
  tr(Number) = tr(901274) ->
  Number = 901274

  tr(Number BETWEEN 900000 AND 909999) -> [pred4]
  (tr(Number) >= tr(900000)) & (tr(Number) <= tr(909999)) ->
  (Number >= 900000) & (Number <= 909999)

  tr(Sport IN 'Tennis', 'Voetbal') -> [pred6] & [pred7]
  (tr(Sport) = tr('Tennis')) | (tr(Sport) = tr('Voetbal')) ->
  (Sport = 'Tennis') | (Sport = 'Voetbal')

  tr(Amount IS NULL) -> [pred14]
  tr(Amount) = null ->
  Amount = null

  tr(Number <= ALL (SELECT Number FROM Students)) -> [pred16]
  for_all <x> @ tr((SELECT Number FROM Students)) : x >= tr(Number) ->
  for_all <x> @ (SELECT Number <= @ Students) : x >= Number

  tr(Number <= ANY (SELECT Number FROM Members)) -> [pred16]
  tr(Number <= SOME (SELECT Number FROM Members)) -> [pred16]
  there_is <x> @ tr((SELECT Number FROM Members)) : x <= tr(Number) ->
  there_is <x> @ (SELECT @ Members <= @ Members) : x <= Number

  tr(Exists SELECT * FROM Members) -> [pred17]
  # tr((SELECT * FROM Members)) ->
  # (@ Members | true)
6 Conclusions

In this paper we have described a formal specification of the database language SQL. Its syntax is given in the corresponding SDF syntax modules. The static semantics of SQL schemas and queries is, at least for a large part, specified by means of an implemented typecheck function. The dynamic semantics of SQL queries which have a meaning in the relational algebra, is described by a translation function from this subset of SQL to a defined and implemented relational database language RDBS.

For this specification of SQL we used ASF+SDF as a formal method instead of the usual definitions in natural language. Thereby the following objectives were kept in mind:

- the relational databases defined and manipulated by the SQL language, are defined using the relational algebra in the same way as we would have done this without keeping SQL in mind, and
- the specification of SQL’s syntax, typecheck rules and semantic translation had to follow the definition given in the standard (BNF, syntax rules, resp. general rules) as close as possible.

The most interesting issues which appeared during this specification process were the following:

- The definition of values and expressions:
  We defined values as the union of sorts describing basic datatypes, and expressions were based on the union of these sorts and variables. Other approaches than the one described here, led to either many undesired ambiguities or a less clear definition of basic datatypes. The major drawback of our approach is that it is time-consuming to incorporate a new datatype.

- Relational algebra instead of relational calculus:
  Most database queries demand logically the evaluation of the Cartesian product of two or more sets. It is very time- and space-consuming to make this evaluation explicit. For this reason, and some others as well, the relational calculus has been defined. This method, which has been proved to have the same power as relational algebra, defines set operations in terms of cursors. Just like SQL cursors, these range over the tuples occurring in some relation (or table).
  We have chosen to implement the relational algebra since this is much more easy than specifying relational calculus, especially using a functional method. To avoid the inefficiency mentioned, we have tried to define the relational operators in terms of
  
  head(RELATION) -> TUPLE and
tail(RELATION) -> RELATION,

  the former giving some tuple of a relation and the latter giving the relation without the same tuple.
  This experiment has been very interesting, but resulted in such an increase in the number of equations that we had to drop it.

- BNF vs. SDF:
  The conversion of BNF to SDF in a straightforward way has been studied briefly to meet the requirement of keeping close to the original syntax definition of SQL in (extended) BNF. On the other hand we defined the relational database language RDBS in the usual way, more like a traditional algebraic specification. See for example the difference in the definition of expressions in both languages (EXPR resp. ValueExpr).
  Afterwards we regret the choice to define the syntax of SQL based on the BNF definition without much modification, since this leads to many unnecessary subsorts and thus to an enormous overhead for the parser. However, subsorts are practical from time to time to reduce the number of cases to be dealt with in the equations.

- Choices made for typechecking:
  - a datatype for the description of an environment (comparable to a symbol table in compilers),
  - simple yet adequate error messages are returned by the typecheck functions,
  - no translation during typecheck is done, but only afterwards and straightforward,
  - views are expanded to base table definitions, after the schema's are typechecked successfully (compare with the problem of expressions in declarations), and
  - passing table lists fulfilling the scope rules for tables and columns in queries (outer references).
  These topics have been discussed in section 4.3 already and will not be repeated here.
• Grouping and set functions:
    As mentioned in the introduction, many claim that SQL is incomprehensive. For a large part this has
to do with the grouping mechanism of SQL and the consequences it has on the allowed results of a
query. For example (table R has columns A and B):
    SELECT A, COUNT(B) FROM R GROUP BY A HAVING COUNT(B) > 0
Without the group by- and having clauses, the SelectList would have been wrong. Due to lack of
time we were not able to incorporate these checks in the typecheck function.

Now for the results of this project. In the first place we consider it a success, but only a partial
success. Indeed it is theoretically and practically possible to define a language like SQL using a
formal method. A designer of an SQL compiler can use such a complete specification as a basis for
his/her design. On the other hand this is probably the maximum gain we get from this method. The
specifications listed in the appendices are not very clear to read (sometimes not even for myself after
some time. I must admit). So we will not claim this specification can be exchanged with the original
standard.

The total amount of work that has been done during my training is not as much as I hoped for
beforehand. There were a number of counterforces, of which we list:

• Specifying a language like SQL top-down by someone who is inexperienced with algebraic
specifications or functional programming, is a bit naive. Working the other way, bottom-up starting
with basic datatypes, expressions, relations and the relational algebra, has proven to be a more
fruitful approach. However implementing the needed datatypes ourselves, instead of using
predefined libraries or hybrid specifications, took too much time.

• The syntax formalism is still a bit low-level in the sense of not having optional- and alternative
constructions in syntax definitions and because the equations require a lot of cases to be written out
in full. When specifying a language like Pascal one may not be aware of this, but SQL has a lot of
these syntax constructions. Therefore changing the syntax or even the name of a variable takes a lot
of time.
7. Personal comments about the ASF+SDF system

In this chapter I would like to express some comments/thoughts concerning the formalism and the system, i.e. the meta-system of ASF+SDF.

First a preliminary remark should be made: the system is not a finished product, it is still evolving; bugs are prepared, problems solved, features added, etc. Even during my training many changes have been made. The most striking example is the additions of the 'hybrid' feature, see [9]. In first instance I left out the datatype REAL from the specification, because this would result in too much work (difficult syntax, many equations, etc). Because of this evolutionary process it is possible that some of my comments are obsolete or premature.

Another remark is that most of my comments are either about syntactic sugar, or about the user interface. In both cases it is not always wanted just to add all the features a user requests.

The remarks are divided into sections, each concerning the sub-system or facility mentioned.

1. Lexical syntax:
   - the formalism lacks grouping of lexical objects:
     when a-d are lexical notions, then so should be: a(bc)*d,
   - the formalism lacks alternatives of lexical objects:
     when a-d are lexical notions, then so is a(bc)d,
   - the formalism lacks optional constructs:
     example: NAT ("." NAT)? ("E" ("+" | "-"))? NAT)? -> REAL
     example (definition of quotes inside a quoted string): ""(""|-""")""* m",

2. Context-free syntax:
   - especially when defining languages it is more convenient to write down and use a BNF grammar than SDF. This has only to do with the direction of a context-free rule. This results in BNF-format in a more readable layout, because there is more layout possible. Looking for a nonterminal in the left margin is more practical in BNF:

   **Example 1**

   **BNF-format**
   
   Statement ::=  
   if <Expression> then <Statements> [ else <Statements> ] fi ;  
   | while <Expression> do <Statements> od ;  
   | begin <Statements> end ;  

   **SDF-format**
   
   "if" Expression "then" Statements "fi" ";"  
   "if" Expression "then" Statements "else" Statements "fi" ";"  
   "while" Expression "do" Statements "od" ";"  
   "begin" Statements "end" ";"  

   So perhaps it is possible to have a context-free function also defined as:
   
   **Id "<-" LexElem+,**
   
   - as can be seen in the example above, also an alternative could be practical,
   - grouping without substot:
   "A (BC)* D -> S" instead of "A T* D -> S  BC -> T",
   - an optional construct would make certain specifications more clear:
   "SELECT("ALL" | "DISTINCT")? SelectList TableExpr -> QuerySpec,
• ambiguities caused by injections of sorts into one another should be resolved using a clear mechanism. For example the expression 4 + 2 has two interpretations (INT "+" INT -> INT or REAL "+" REAL -> REAL) when an integer constant is defined as a real constant as well (e.g. INT -> REAL, INT "." INT -> REAL). I would prefer this expression to be treated as an INT expression by default and as a REAL expression when the priority REAL "+" REAL -> REAL > INT "+" INT -> INT (or REAL > INT) is given in the priorities section.

3. Modularity:
• it should be possible to restrict imported and especially exported signatures,
• it should be possible to split up the equations logically belonging to a single syntax specification, into two or more equations sections; combining semantics to syntax should preferably be done in the imports specification. Sometimes only a part of the syntax c.q. semantics is needed. In the first case we can restrict the imported syntax using restricted imports, the latter should be possible too,
• the name of a module should be part of the module syntax; using the name of the file on which a module can be found as module name, is not an elegant choice,
• empty sections should be allowed; it is annoying to have to remove 'lexical syntax' when there are no lexical functions or to remove 'equations' when we want to comment out some equations temporary,
• cyclic imports caused by dividing a specification into two or more modules should be possible.

4. Meta-system:
• mapping of modules (syntax plus semantics) to filenames and vice versa, should be a specification of its own,
• asking a user to save changes made to a text should be done before the window is closed.

5. Variables:
• the flexible declaration of variables is excellent.

6. Equations:
• I don’t like tags, I dislike their syntax,
• meta-syntax for equations can be defined to ease the problem of dealing with many cases, e.g. some sort of case construct for the result of an equation:

Example 2

    _Expr1 + _Expr2 =
    CASE
      type(_Expr1) = integer & type(_Expr2) = integer  ===> add-integer(_Expr1, _Expr2)
      type(_Expr1) = integer & type(_Expr2) = real    ===> add-real(_Expr1, _Expr2)
      type(_Expr1) = real & type(_Expr2) = integer    ===> add-real(_Expr1, _Expr2)
      type(_Expr1) = real & type(_Expr2) = real       ===> add-real(_Expr1, _Expr2)
      DEFAULT                          ===> ERROR!
    END

• the order of equations tried to reduce a term should either be defined (first rule first, lowest tag first, or whatever) or intentionally random. The situation as it is now (first rule first, but not guaranteed) is too tempting for not anticipating on the order of evaluation,
• many language constructs consist of lists (or optionals), so in many situations we have to deal with the cases empty vs. non-empty. An empty list (or optional) is a syntactically difficult object, because it is literally empty. The usual solution is to surround a list by opening- and closing brackets (e.g. \{\} or \begin/end). Especially for an optional defined as a subpart of its own, it should be possible to express the case that the optional is empty, as given in the following example:

Example 3

    "procedure" OptArgList ";"   --> ProcDecl
    "(" ArgList ")"              --> OptArgList
    % empty %                   --> OptArgList
• the choice for the symbol "−" to indicate the fact that left-hand side equals right-hand side in an equation, is obvious for logical and historical reasons; it should be possible however to use another symbol (like "!−!") when we want to specify a function S "−" S → BOOL of our own,
• in many cases it is convenient to consider conditions as boolean conditions. I do not suggest that booleans should be incorporated in the ASF+SDF formalism. However, an operator resembling boolean "or" (just like ",," resembles much to boolean "and") would be a nice-to-have feature as can be seen in the following example:

Example 4

[..] type(e1) = real | type(e2) = real => e1 + e2 = real-addition(e1, e2)

7. Reductions:
• also as a debugging facility it would be nice to have a reduction of the contents of the focus alone, e.g. tc(descr(schemata), query) → reduce→ tc(canvysdesc, query); preferably in the same window (so replacing the subterm in the focus by its normal form),
• another useful feature would be the possibility to have the result of a reduction be written in a separate output window; an input window for 'user' entries would likewise be thrilling.

8. Debugger:
The debug facility EDB, as described in [12], has been used frequently. A later version of this debugger was not quite suited for my application (because of the very large terms), so I used it less than before. I personally would prefer to have a window containing a term to be reduced, a selection of a subterm to be rewritten next (with a message which equation is going to be used) and a set of menus and buttons for controlling this rewriting. After the, say, 'step'-button, the selected subterm is then to be replaced by another term. So after repeatedly pressing this button, or otherwise, the total rewriting process can be simulated visually. The level of detail of the rewriting should be limited, for instance: show reductions of the current module in detail and only show the results of reductions of other modules. Testing conditions can be done in newly created windows, depending on some level set by the user. E.g. some level only shows reductions (and eventually success or failure of conditions) and yet another level shows reductions of a condition in another window.

9. Editor:
• using the power of ASF+SDF it would be nice to have a find/replace facility, for instance: change the name of the function INT + INT → INT to function int-add(IN, INT) → INT. Applying this replace to a context-free function definition, this could result in changing some or all equations where this function occurs,
• the permanent display of the sortname of current focus, instead of using the expand menu, would be practical for inspecting terms in a language,
• it should be possible to position the cursor using a mouseclick without parsing,
• it is understandable, but still annoying, that zooming out should lead to a parse.

10. Miscellaneous:
• pretty print: too soon is chosen for a vertical layout for list structures; large terms (like a RDBS program for instance) become soon unreadable,
• using the disambiguator (dialog window showing the possible parse trees of a subterm) should be avoided as much as possible (there must be at least a possibility to interrupt the current parse). I would like to have a default choice mechanism for each ambiguity possible; this method, and the way to overrule it, should be obvious to the average user.
8 References


Appendix A: BNF grammar definition of SQL

5. Common elements

5.1 <character>
<character> ::= <digit> | <letter> | <special character>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> ::= <upper case letter> | <lower case letter>
<upper case letter> ::= A | B | C | D | E | F | G | H | I
| J | K | L | M | N | O | P | Q | R
| S | T | U | V | W | X | Y | Z
<lower case letter> ::= a | b | c | d | e | f | g | h | i
| j | k | l | m | n | o | p | q | r
| s | t | u | v | w | x | y | z
<special character> ::= 
  See Syntax Rule 1.

5.2 <literal>
<literal> ::= <character string literal> | <numeric literal>
<character string literal> ::= ' <character representation> ... '
<character representation> ::= <nonquote character> | <quote representation>
<nonquote character> ::= See Syntax Rule 1.
<quote representation> ::= "
<numeric literal> ::= <exact numeric literal> | <approximate numeric literal>
<exact numeric literal> ::= [+|-] <unsigned integer>[. <unsigned integer>] |
  [+|-] <unsigned integer> . |
  [+|-] . <unsigned integer>
<approximate numeric literal> ::= <mantissa>E<exponent>
<mantissa> ::= <exact numeric literal>
<exponent> ::= <signed integer>
<signed integer> ::= [+|-]<unsigned integer>
<unsigned integer> ::= <digit>...

5.3 <token>
<token> ::= <nondelimiter token> | <delimiter token>
<nondelimiter token> ::= <identifier> | <keyword> | <numeric literal>
<identifier> ::= <upper case letter> | [[<underscore>|<letter or digit>]]...
<underscore> ::= _
<letter or digit> ::= <upper case letter> | <digit>
<keyword> ::= ALL | AND | ANY | AS | ASC | AUTHORIZATION | AVG
| BEGIN | BETWEEN | BY
| CHAR | CHARACTER | CHECK | CLOSE | COBOL | COMMIT
| CONTINUE | COUNT | CREATE | CURRENT | CURSOR
| DEC | DECIMAL | DECLARE | DEFAULT | DELETE | DESC | DISTINCT | DOUBLE
| END | ESCAPE | EXEC | EXISTS
| FETCH | FLOAT | FOR | FOREIGN | FORTRAN | FOUND | FROM
| GO | GOTO | GRANT | GROUP | HAVING |
| IN | INDICATOR | INSERT | INT | INTEGER | INTO | IS
| KEY | LANGUAGE | LIKE
| MAX | MIN | MODULE | NOT | NULL | NUMERIC
| OF | ON | OPEN | OPTION | OR | ORDER
| PASCAL | PL1 | PRECISION | PRIMARY | PRIVILEGES | PROCEDURE | PUBLIC
| REAL | REFERENCES | ROLLBACK
| SCHEMA | SECTION | SELECT | SET | SMALLINT | SOME
| SQL | SQLCODE | SQLERROR | SUM
| TABLE | TO | UNION | UNIQUE | UPDATE | USER,
| VALUES | VIEW | WHENEVER | WHERE | WITH | WORK
5.4 Names
<table name> ::= [<authorization identifier>.]<table identifier>
<authorization identifier> ::= <identifier>
<table identifier> ::= <identifier>
<column name> ::= <identifier>
<correlation name> ::= <identifier>
<module name> ::= <identifier>
<cursor name> ::= <identifier>
<procedure name> ::= <identifier>
<parameter name> ::= <identifier>

5.5 <data type>
<data type> ::= 
    <character string type>
    | <exact numeric type>
    | <approximate numeric type>
<character string type> ::= CHARACTER[<length>] | CEAR[<length>]
<exact numeric type> ::= 
    NUMERIC[<precision>,<scale>]
    | DECIMAL[<precision>,<scale>]
    | DEC[<precision>,<scale>]
    | INTEGER
    | INT
    | SMALLINT
<approximate numeric type> ::= 
    REAL
    | DOUBLE PRECISION
<precision ::= <unsigned integer>
<length ::= <unsigned integer>
<scale ::= <unsigned integer>

5.6 <value specification> and <target specification>
<value specification> ::= 
    <parameter specification>
    | <variable specification>
    | <literal>
    | USER
<target specification> ::= <parameter specification> | <variable specification>
<parameter specification> ::= <parameter name>[<indicator parameter>]
<indicator parameter> ::= [INDICATOR]<parameter name>
<variable specification> ::= <embedded variable name>[<indicator variable>]
<indicator variable> ::= [INDICATOR]<embedded variable name>

5.7 <column specification>
<column specification> ::= [<qualifier>.]<column name>
<qualifier> ::= <table name> | <correlation name>
5.8 <set function specification>
<set function specification> ::= 
  COUNT(*) 
  | <distinct set function> 
  | <all set function> 
<distinct set function> ::= 
  { AVG | MAX | MIN | SUM | COUNT}(DISTINCT <column specification>)
<all set function> ::= 
  { AVG | MAX | MIN | SUM}([ALL] <value expression>)

5.8 <value expression>
<value expression> ::= 
  <term> 
  | <value expression> + <term> 
  | <value expression> - <term> 
<term> ::= 
  <factor> 
  | <term> * <factor> 
  | <term> / <factor> 
<factor> ::= [+|-]<primary> 
<primary> ::= 
  <value specification> 
  | <column specification> 
  | <set function specification> 
  | (<value expression>)

5.10 <predicate>
<predicate> ::= 
  <comparison predicate> 
  | <between predicate> 
  | <in predicate> 
  | <like predicate> 
  | <null predicate> 
  | <quantified predicate> 
  | <exists predicate> 

5.11 <comparison predicate>
<comparison predicate> ::= 
  <value expression> <comp op> [<value expression> | <subquery>] 
<comp op> ::= = | < | <= | > | >= |

5.12 <between predicate>
<between predicate> ::= 
  <value expression> [NOT] BETWEEN <value expression> AND <value expression>

5.13 <in predicate>
<in predicate> ::= 
  <value expression> [NOT] IN {<subquery> | (<in value list>)} 
<in value list> ::= 
  <value specification> {,<value specification>}

5.14 <like predicate>
<like predicate> ::= 
  <column specification> [NOT] LIKE <pattern> [ESCAPE <escape character>] 
<pattern> ::= <value specification> 
<escape character> ::= <value specification>

5.15 <null predicate>
<null predicate> ::= <column specification> IS [NOT] NULL
5.16 <quantified predicate>
<quantified predicate> ::= <value expression> <comp op> <quantifier> <subquery>
<quantifier> ::= <all> | <some>
<all> ::= ALL
<some> ::= SOME | ANY

5.17 <exists predicate>
<exists predicate> ::= EXISTS <subquery>

5.18 <search condition>
<search condition> ::= 
  <boolean term>
  | <search condition> OR <boolean term>
<boolean term> ::= 
  <boolean factor>
  | <boolean term> AND <boolean factor>
<boolean factor> ::= [NOT] <boolean primary>
<boolean primary> ::= <predicate> | (<search condition>)

5.19 <table expression>
<table expression> ::= 
  <from clause>
  [<where clause>]
  [<group by clause>]
  [<having clause>]

5.20 <from clause>
<from clause> ::= FROM <table reference>[,{,<table reference>}]...
<table reference> ::= <table name> [<correlation name>

5.21 <where clause>
<where clause> ::= WHERE <search condition>

5.22 <group by clause>
<group by clause> ::= 
  GROUP BY <column specification>[,{,<column specification>}]...

5.23 <having clause>
<having clause> ::= HAVING <search condition>

5.24 <subquery>
<subquery> ::= (SELECT [ALL|DISTINCT] <result specification><table expression>)
[result specification] ::= <value expression> | *

5.25 <query specification>
<query specification> ::= SELECT [ALL|DISTINCT] <select list><table expression>
<select list> ::= <value expression>[,{,<value expression>}] | *

6. Schema definition language

6.1 <schema>
/schema> ::= 
  CREATE SCHEMA <schema authorization clause> [<schema element>...]
/schema authorization clause> ::= 
  AUTHORIZATION <schema authorization identifier>
/schema authorization identifier> ::= <authorization identifier>
/schema element> ::= 
  <table definition>
  | <view definition>
  | <privilege definition>
6.2 <table definition>
<table definition> ::= CREATE TABLE <table name> [<table element>[,(<table element>)]...]
<table element> ::= <column definition>
| <table constraint definition>

6.3 <column definition>
<column definition> ::= <column name> <data type> [default clause] [column constraint]...
<column constraint> ::= NOT NULL <uniqueness specification>
| <referential constraint definition>
| CHECK(<search condition>)

6.4 <default clause>
<default clause> ::= DEFAULT {<literal> | USER | NULL}

6.5 <table constraint definition>
<table constraint definition> ::= <uniqueness specification>
| <referential constraint definition>
| <check constraint definition>

6.6 <uniqueness specification>
<uniqueness specification> ::= <unique column list>
<unique specification> ::= UNIQUE | PRIMARY KEY
<unique column list> ::= <column name>[,(<column name>)]...

6.7 <referential constraint definition>
<referential constraint definition> ::= FOREIGN KEY(<referencing columns>)
<references specification> ::= REFERENCES <referenced table and columns>
<referencing columns> ::= <reference column list>
<referenced table and columns> ::= <table name>[<reference column list>]...
<reference column list> ::= <column name>[,(<column name>)]...

6.8 <check constraint definition>
<check constraint definition> ::= CHECK(<search condition>)

6.9 <view definition>
<view definition> ::= CREATE VIEW <table name> [(<view column list>)]
AS <query specification> [WITH CHECK OPTION]

6.10 <privilege definition>
<privilege definition> ::= GRANT <privileges> ON <table name>
TO <grantee>[,<grantee>...]
[WITH CHECK OPTION]
<privileges> ::= ALL PRIVILEGES
| <action>[,<action>]...
$action> ::= SELECT | INSERT | DELETE
| UPDATE [(<grant column list>)]
| REFERENCES [(<grant column list>)]
<grant column list> ::= <column name>[,(<column name>)]
<grantee> ::= PUBLIC | <authorization identifier>
7. Module language

7.1 <module>
<module> ::=<module name clause>
    <language clause>
    <module authorization clause>
    [<declare cursor>...]
    <procedure>...

7.2 <module name clause>
<module name clause> ::= MODULE [<module name>]

7.3 <procedure>
<procedure> ::=PROCEDURE <procedure name> <parameter declaration>...;
<SQL statement>;
<parameter declaration> ::=<parameter name> <datatype>
    | <SQLCODE parameter>
<SQLCODE parameter> ::=<SQLCODE
<SQL statement> ::=<close statement>
    | <commit statement>
    | <delete statement: positioned>
    | <delete statement: searched>
    | <fetch statement>
    | <insert statement>
    | <open statement>
    | <rollback statement>
    | <select statement>
    | <update statement: positioned>
    | <update statement: searched>

8. Data manipulation language

8.1 <close statement>
<close statement> ::= CLOSE <cursor name>

8.2 <commit statement>
<commit statement> ::= COMMIT WORK

8.3 <declare cursor>
<declare cursor> ::=DECLARE <cursor name> CURSOR FOR <cursor specification>
<cursor specification> ::=<query expression>[<order by clause>]
<query expression> ::=<query term>
    | <query expression> UNION [ALL] <query term>
<query term> ::=<query specification> | (<query expression>)
<order by clause> ::=ORDER BY <sort specification> [{, <sort specification>}...]
<sort specification> ::=<{unsigned integer} | <column specification>} [ASC | DESC]

8.4 <delete statement: positioned>
<delete statement: positioned> ::=DELETE FROM <table name> WHERE CURRENT OF <cursor name>
8.5 <delete statement: searched>
<delete statement: searched> ::= 
  DELETE FROM <table name> [WHERE <search condition>]

8.6 <fetch statement>
<fetch statement> ::= 
  FETCH <cursor name> INTO <fetch target list>
<fetch target list> ::= 
  <target specification> [{, <target specification>}...]

8.7 <insert statement>
<insert statement> ::= 
  INSERT INTO <table name> [{<insert column list>}]
  { VALUES {<insert value list>}|<query specification>}
<insert column list> ::= 
  <column name> [{, <column name>}...]
<insert value list> ::= 
  <insert value> [{, <insert value>}...]
<insert value> ::= 
  <value specification> | NULL

8.8 <open statement>
<open statement> ::= OPEN <cursor name>

8.9 <rollback statement>
rollback statement ::= ROLLBACK WORK

8.10 <select statement>
<select statement> ::= 
  SELECT [ALL | DISTINCT] <select list>
  INTO <select target list>
  <table expression>
<select target list> ::= 
  <target specification> [{, <target specification>}...]

8.11 <update statement: positioned>
<update statement: positioned> ::= 
  UPDATE <table name>
  SET <set clause: positioned> [{, <set clause: positioned>}...]
  WHERE CURRENT OF <cursor name>
<set clause: positioned> ::= 
  <object column: positioned> = {<value expression> | NULL}
<object column: positioned> ::= <column name>

8.12 <update statement: searched>
<update statement: searched> ::= 
  UPDATE <table name>
  SET <set clause: searched> [{, <set clause: searched>}...]
  [WHERE <search condition>]
<set clause: searched> ::= 
  <object column: searched> = {<value expression> | NULL}
<object column: searched> ::= <column name>
module Layout

exports

lexical syntax
[ \
]  -> LAYOUT
"\n" -\[\n]\* \[\n]  -> LAYOUT
"\n" -\{\n\}+ "\n"  -> LAYOUT

variables
". Char*[0-9]*"  -> CHAR
". Char*[0-9]*"  -> CHAR
". Char*[0-9]*"  -> CHAR

end Layout

module Lisp

%%
%% The module LISP
%%
%% The KEM knows about this module.
%% DON'T EDIT, except when you want to add Lispids
%% to the lexical (not context-free) syntax.
%%
%% Not yet supported:
%% ' and ,

exports

sorts
LISP
  % hidden
  LispAtom StringPart PackagePart QuotedLispAtom LispId LispString

lexical syntax
%% All non control or meta characters that have characterclass pname.
%% \{\$#%+\-\/=\<\?\@\-\$\}\s-\}+  -> LispAtom
\[a-zA-Z0-9\-\}+ \}  -> LispAtom
"\*" LispAtom  -> PackagePart
"\*" PackagePart+  -> LispId
"\["[0-9]+  -> LispId
"\["[0-9a-zA-Z]+  -> LispId
"\["[a-zA-Z-\}+]  -> LispId
"\["[[]]+ "\}"  -> QuotedLispAtom
QuotedLispAtom+  -> LispAtom
LispAtom  -> LispId
\[\n]  -> LispId
\[\n]  -> StringPart
StringPart+  -> LispString
[ \
]  -> LAYOUT

context-free syntax
LispId  -> LISP
LispString  -> LISP
"(" LISP ")"  -> LISP
%% And when you prefer them, they aren't preferred.
"<<" Condition ")>>  -> LISP
"'" LISP  -> LISP

end Lisp
module Booleans

imports

Layout

exports

sorts

BOOL_CONST
BOOL

lexical syntax

"true" -> BOOL_CONST
"false" -> BOOL_CONST

context-free syntax

BOOL_CONST
"!" BOOL
BOOL "[" BOOL
BOOL "]" BOOL
BOOL "=" BOOL
BOOL "<>" BOOL

variables

"_BoolConst"[0-9]* -> BOOL_CONST
"_Bool"[0-9]* -> BOOL

priorities

"!" BOOL -> BOOL
BOOL "[" BOOL -> BOOL, BOOL "|"]" BOOL -> BOOL
BOOL "=" BOOL -> BOOL, BOOL "<>" BOOL -> BOOL

equations

[not1] !true = false
[not2] !false = true
[or1] true | _Bool = true
[or2] false | _Bool = _Bool
[or3] _Bool | true = true
[or4] _Bool | false = _Bool
[and1] true & _Bool = _Bool
[and2] false & _Bool = false
[and3] _Bool & true = _Bool
[and4] _Bool & false = false

[eql1] _BoolConst1 = _BoolConst2 ->
(_BoolConst1 = _BoolConst2) = true
[eql2] _BoolConst1 = _BoolConst2 ->
(_BoolConst1 = _BoolConst2) = false
[neq1] _BoolConst1 = _BoolConst2 ->
(_BoolConst1 <> _BoolConst2) = true
[neq2] _BoolConst1 = _BoolConst2 ->
(_BoolConst1 <> _BoolConst2) = false

end Booleans
[plus6] int const (Char) + int const ("s") = [ 6, 7, 8, 9,10,11,12,13,14,15].digit (Char)
[plus7] int const (Char) + int const ("t") = [ 7, 8, 9,10,11,12,13,14,15,16].digit (Char)
[plus8] int const (Char) + int const ("k") = [ 8, 9,10,11,12,13,14,15,16,17].digit (Char)
[plus9] int const (Char) + int const ("n") = [ 9,10,11,12,13,14,15,16,17,18].digit (Char)

[plus10] int const (Char+1 Char1) + int const (Char+2 Char2) =
  int const (Char+ Char)
  when int const (Char1) + int const (Char2) = int const (Char+ Char),
  int const (Char+1) + int const (Char+2) + int const ("0" Char*) =
  int const (Char+1 Char1)

[plus11] int const (Char+1 Char1) + int const (Char+2 Char2) =
  int const (Char+ Char)
  when int const (Char1) + int const (Char2) = int const (Char+ Char),
  int const (Char+1) + int const ("0" Char*) =
  int const (Char+1 Char1)

[plus12] int const (Char+1 Char1) + int const (Char+2 Char2) =
  int const (Char+ Char)
  when int const (Char1) + int const (Char2) = int const (Char+ Char),
  int const (Char+2) + int const ("0" Char*) =
  int const (Char+2 Char2)

[plus13] _IntConst1 + _IntConst1 = (_IntConst1 _IntConst1)
[plus14] _IntConst1 + _IntConst2 = (_IntConst2 _IntConst1)
[plus15] _IntConst1 + _IntConst2 = (_IntConst1 _IntConst2)

%% INT -> INT (subtraction)

[min0] _IntConst1 - _IntConst2 = (_IntConst1 _IntConst2)
  when _IntConst1 > _IntConst2 = true
[min1] _IntConst1 - _IntConst2 = (_IntConst2 _IntConst1)
  when _IntConst1 > _IntConst2 = false
[min2] _IntConst1 - _IntConst2 = (_IntConst1 + _IntConst2)
[min3] _IntConst1 - _IntConst2 = (_IntConst2 + _IntConst1)
[min4] _IntConst1 - _IntConst2 = (_IntConst2 - _IntConst1)

[min0] int const (Char) _int const ("0") = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9].digit (Char)
[min1] int const (Char) _int const ("1") = [19, 0, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)
[min2] int const (Char) _int const ("2") = [19, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)
[min3] int const (Char) _int const ("3") = [19, 1, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)
[min4] int const (Char) _int const ("4") = [19, 0, 1, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)
[min5] int const (Char) _int const ("5") = [19, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)
[min6] int const (Char) _int const ("6") = [19, 2, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)
[min7] int const (Char) _int const ("7") = [19, 3, 0, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)
[min8] int const (Char) _int const ("8") = [19, 3, 1, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)
[min9] int const (Char) _int const ("9") = [19, 4, 0, 1, 2, 3, 4, 5, 6, 7, 8].digit (Char)

[min10] int const (Char1 Char1) _int const (Char2 Char2) =
  int const (Char Char)
  when int const (Char1) + int const (Char2) = int const (Char+ Char),
  int const (Char+1) _int const (Char+2) = int const (Char+ Char)
  int const (Char Char)
  when int const (Char1) + int const (Char2) = int const (Char+ Char),
  int const (Char+1) _int const ("0" Char*) =
  int const (Char+ Char)

[min11] int const (Char1 Char1) _int const (Char2 Char2) =
  int const (Char Char)
  when int const (Char1) + int const (Char2) = int const (Char+ Char),
  int const (Char+1) _int const ("0" Char*) =
  int const (Char+ Char)

%% INT * INT (multiplication)

[time0] int const (Char) * int const ("0") = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0].digit (Char)
[time1] int const (Char) * int const ("1") = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9].digit (Char)
[time2] int const (Char) * int const ("2") = [ 0, 2, 4, 6, 8,10,12,14,16,18].digit (Char)
[time3] int const (Char) * int const ("3") = [ 0, 3, 6, 9,12,15,18,21,24,27].digit (Char)
[time4] int const (Char) * int const ("4") = [ 0, 4, 8,12,16,20,24,28,32,36].digit (Char)
[time5] int const (Char) * int const ("5") = [ 0, 5,10,15,20,25,30,35,40,45].digit (Char)
[time6] int const (Char) * int const ("6") = [ 0, 6,12,18,24,30,36,42,48,54].digit (Char)
[time7] int const (Char) * int const ("7") = [ 0, 7,14,21,28,35,42,49,56,63].digit (Char)
[time8] int const (Char) * int const ("8") = [ 0, 8,16,24,32,40,48,56,64,72].digit (Char)
[time9] int const (Char) * int const ("9") = [ 0, 9,18,27,36,45,54,63,72,81].digit (Char)

[time10] int * int const ("0") = 0
[time11] int * int const (Char) =
  int + int * ([0,1,2,3,4,5,6,7,8]).digit (Char)
  when int const (Char) = int const ("0")
[time12] int const (Char1) * int const (Char2 Char2) =
  int const (Char+ Char) + int const (Char1) * int const (Char2 Char2) - int const (Char)
  when int const (Char1) * int const (Char2 Char2) = int const (Char)

[time13] _IntConst1 = _IntConst2 = (_IntConst1 _IntConst2)
[time14] _IntConst1 _IntConst2 = (_IntConst1 _IntConst2)
[time15] _IntConst1 _IntConst2 = (_IntConst1 _IntConst2)

%% INT / INT (integer division)

div1 _IntConst1 _IntConst2 = (_IntConst1 _IntConst2)
div2 _IntConst1 _IntConst2 = (_IntConst1 _IntConst2)
div3 _IntConst1 _IntConst2 = (_IntConst1 _IntConst2)
div4 _IntConst1 _IntConst2 = 0
  when _IntConst2 > _IntConst1 = true
**INT 0 INT (modulo)**

when \(\_\_\text{IntConst}1 > \_\_\text{IntConst}2\) = true

**INT 0 INT (equality)**

when \(\_\_\text{IntConst}1 = \_\_\text{IntConst}2\) = true

**INT 0 INT (inequality)**

when \(\_\_\text{IntConst}1 < \_\_\text{IntConst}2\) = true

**INT 0 INT (greater than)**

when \(\_\_\text{IntConst}1 > \_\_\text{IntConst}2\) = true

**INT 0 INT (less than)**

when \(\_\_\text{IntConst}1 < \_\_\text{IntConst}2\) = true

end Integers
module Reals

imports

Layout;
Lisp;
Booleans;
Integers;

exports

sorts

REAL_CONST
REAL

lexical syntax

MANTISSA -> REAL_CONST
MANTISSA EXPONENT -> REAL_CONST
[Int Const] -> EXPONENT
["+" Int Const] -> EXPONENT
["-" Int Const] -> EXPONENT
INT_CONST "," INT CONST -> MANTISSA
INT_CONST "+" INT CONST -> MANTISSA
INT_CONST "-" INT CONST -> MANTISSA

context-free syntax

REAL_CONST -> REAL
"+" REAL -> REAL
"-" REAL -> REAL (left)
REAL "+" REAL -> REAL (non-associative)
REAL "-" REAL -> REAL (left)
REAL "/" REAL -> REAL (non-associative)
"(" REAL ")" -> REAL (bracket)

REAL "+=" REAL -> BOOL
REAL "+=" REAL -> BOOL
REAL "+" REAL -> BOOL
REAL "<" REAL -> BOOL
REAL ">=" REAL -> BOOL
REAL "<" REAL -> BOOL
REAL max (" REAL "," REAL ")" -> REAL
REAL min (" REAL "," REAL ")" -> REAL

variables

"RealConst" [0-9]* -> REAL_CONST
"Real" [0-9]* -> REAL

priorities

{ "+" REAL -> REAL, "+=" REAL -> REAL } >
{ left: REAL "+=" REAL -> REAL, REAL "/" REAL -> REAL } >
{ right: REAL "+=" REAL -> REAL, REAL "-" REAL -> REAL } >
{ REAL "=" REAL -> BOOL, REAL "<" REAL -> BOOL, REAL ">=" REAL -> BOOL, REAL "+" REAL -> BOOL, REAL "-" REAL -> BOOL } >

hiddens

sorts

REAL

context-free syntax

"LISP" LISP -> REAL
"REAL" LISP -> REAL
"<<" REAL ">=" -> LISP
"++" -> LISP
"-++" -> LISP
"*/" -> LISP
"max" -> LISP
"min" -> LISP

equations

[uplus] _ Real = _ Real
[umin] _ Real = _ Real
[add] _ RealConst1 + _ RealConst2 = REAL (convert-to-lexical "<< 1.0 >>")
(- (convert-lexical-to-lisp "<< _RealConst1 >>")
module Char

imports

Layout
Lisp
Booleans
Integers

exports

sorts

CHR

lexical syntax

`[-] "="  ->  CHR
"="     ->  CHR

context-free syntax

CHR "=" CHR  ->  BOOL
CHR "<" CHR  ->  BOOL
CHR "<=" CHR ->  BOOL
CHR ">=" CHR ->  BOOL
CHR "<>" CHR  ->  BOOL
CHR "<" CHR  ->  BOOL
CHR "<=" CHR  ->  BOOL

variables

"_Chr"[0-9]*  ->  CHR

hiddens

context-free syntax

(ord "(" CHR ")"  ->  INT
"INT" Lisp  ->  INT
"<<" INT ">>"  ->  IISP
"<<" CHR ">>"  ->  IISP

equations

[ 1] ord(_Chr) = INT [convert-to-lexical
   << 1 >>
   (ascii (convert-lexical-to-lisp << _chr >>)))

[ 2] _Chr = _Chr = true
[ 3] _Chr1 = _Chr2 = false
   when _Chr1 != _Chr2
[ 4] _Chr1 <> _Chr2 = !(_Chr1 = _Chr2)
[ 5] _Chr1 > _Chr2 = ord(_Chr1) > ord(_Chr2)
[ 6] _Chr1 < _Chr2 = ord(_Chr1) < ord(_Chr2)
[ 7] _Chr1 >= _Chr2 = ord(_Chr1) >= ord(_Chr2)
[ 8] _Chr1 <= _Chr2 = ord(_Chr1) <= ord(_Chr2)

end Char
module Strings

imports

Layout:
Booleans
Integers
Chars

exports

sorts

STR_CONST
STR

lexical syntax

"" 
"[" 
"*" 
"+" 
"-" 
"!" 
"<" 
">" 
">=" 
"<=" 
"@" 
"=" 
"<" 
">" 
"*" 
"*" 

-> STR_CONST

context-free syntax

STR_CONST

-> STR

STR "*" STR

-> STR { left }

INT "*" STR

-> STR

STR "=" STR

-> BOOL

STR "<" STR

-> BOOL

STR ">" STR

-> BOOL

STR "@" STR

-> BOOL

length "(" STR ")"

-> INT

variables

"_StrConst*[0-9]*" 

-> STR_CONST

"_Str"[0-9]*

-> STR

priorities

STR "*" STR -> STR 
( STR "=" STR -> BOOL, STR "<" STR -> BOOL )

equations

[cat5] str_const("" Char1 "") + str_const("" _Char*2 "") = 
str_const("" _Char1 _Char*2 "")

[rep1] 0 * _Str = ''

[rep2] _Int > 0 = true => 
Int * _Str = _Str + dec(Int) * _Str

[eql] _StrConst1 = _StrConst2 => 
_StrConst1 = _StrConst2 = true

[eql] _StrConst1 /= _StrConst2 =>
_StrConst1 = _StrConst2 = false

[neq1] _StrConst1 <= _StrConst2 = 
((_StrConst1 <= _StrConst2)

[geq1] _Str < _Str1 = _Str2 > _Str1

[geq1] _Str1 > _Str2 = (_Str1 > _Str2) | ( _Str1 = _Str2)

[leq1] _Str <= _Str2 = ( _Str1 > _Str2) | ( _Str1 = _Str2)

[leq0] length(str_const("" "")) = 0

[leq1] length(str_const("" _Char" "")) = 
len(length(str_const("" _Char" "")))

end Strings
module Values

imports
Layout
Booleans
Integers
Reals
Strings

exports

sorts

VAL

lexical syntax

"null" -> VAL

context-free syntax

BOOL -> VAL
INT -> VAL
REAL -> VAL
STR

itor "(" INT ")" -> REAL
btos "(" BOOL ")" -> STR
itos "(" INT ")" -> STR
rtos "(" REAL ")" -> STR

variables

"_Val"[0-9]*
"_Val"[0-9]*

equations

[itor1] itor(int_const(Char+)) = real_const(Char+ ".")
[itor2] itor(- _Int) = - itor(_Int)

[btos1] btos(true) = "true"
[btos2] btos(false) = "false"

[itos1] itos(int_const(Char+)) = str_const("" _Char+ "")
[itos2] itos(- _Int) = "-" + itos(_Int)

[rtos1] rtos(real_const(Char+)) = str_const("" _Char+ "")
[rtos2] rtos(- _Real) = "-" + rtos(_Real)

end Values

module Variables

imports
Layout

exports

sorts

VAR

lexical syntax

[A-Za-z][A-Za-z0-9]*

variables

"_Var"[0-9]*
"_Var"[0-9]*

end Variables
module Expressions

imports

Layout
Values
Variables

exports

sorts

EXPR

context-free syntax

BOOL → EXPR
INT → EXPR
REAL → EXPR
STR → EXPR
VAR → EXPR

"!" EXPR → EXPR
"=" EXPR → EXPR

EXPR "+" EXPR → EXPR (left)
EXPR "-" EXPR → EXPR (left)
EXPR "/" EXPR → EXPR (left)

EXPR "<" EXPR → EXPR (left)
EXPR "<=" EXPR → EXPR (left)
EXPR "<" EXPR → EXPR (left)
EXPR "<=" EXPR → EXPR (left)

EXPR "/" EXPR → EXPR (left)
EXPR "|" EXPR → EXPR (left)

"(" EXPR ")" → EXPR (bracket)

"(" EXPR ")" → EXPR
"(" EXPR ")" → EXPR
"(" EXPR ")" → EXPR

bool "(" BOOL ")" → BOOL
int "(" INT ")" → INT
real "(" REAL ")" → REAL
str "(" STR ")" → STR
val "(" EXPR ")" → VAL

variables

"_Expr[0-9]*" → EXPR
"_Exprs[0-9]*" → EXPR

priorities

{"!" EXPR → EXPR, "!" EXPR → EXPR, "=" EXPR → EXPR};
{left: EXPR "*" EXPR → EXPR, EXPR "/" EXPR → EXPR};
{left: EXPR "<" EXPR → EXPR, EXPR "<=" EXPR → EXPR, EXPR "<=" EXPR → EXPR, EXPR ">=" EXPR → EXPR, EXPR ">=" EXPR → EXPR, EXPR "<=" EXPR → EXPR};
{left: EXPR ":=" EXPR → EXPR, EXPR "|" EXPR → EXPR}

equations

[force1] bool(_Bool) = _Bool
[force2] int(_Int) = _Int
[force3] real(_Real) = _Real
[force4] str(_Str) = _Str
[val1] val(_Bool) = _Bool
[val2] val(_Int) = _Int
[val3] val(_Real) = _Real
[val4] val(_Str) = _Str
[not1] ! _Expr = bool(!_Bool) when _Expr = _Bool
[not2] !_ Expr = null when _Expr = _Int
[not3] !_ Expr = null when _Expr = _Real
[not4] !_ Expr = null when _Expr = _Str
[not5] !_ Expr = null when _Expr = null
[plus1] + _Expr = int(_Int) when _Expr = _Int
[plus2] + _Expr = real(_Real) when _Expr = _Real
[um1] = _Expr = int(_Int) when _Expr = Int
[um2] = _Expr = real(_Real) when _Expr = Real
[add1] _Expr1 + _Expr2 = int(_Int1 + _Int2) when _Expr1 = Int, _Expr2 = Int
[add2] _Expr1 + _Expr2 = real(_Real1 + _Real2) when _Expr1 = Real, _Expr2 = Real
[add3] _Expr1 + _Expr2 = real(_Real1 + itor(_Int)) when _Expr1 = Real, _Expr2 = Int
[add4] _Expr1 + _Expr2 = real(itor(_Int) + _Real) when _Expr1 = Int, _Expr2 = Real
[add5] _Expr1 + _Expr2 = str(_Str1 + _Str2) when _Expr1 = str, _Expr2 = str

**[add6]** default: _Expr1 + _Expr2 = null

[add7] _Expr1 + _Expr2 = null when _Expr1 = Bool, _Expr2 = Bool
[add8] _Expr1 + _Expr2 = null when _Expr1 = Bool, _Expr2 = Int
[add9] _Expr1 + _Expr2 = null when _Expr1 = Bool, _Expr2 = Str
[add10] _Expr1 + _Expr2 = null when _Expr1 = Bool, _Expr2 = null
[add11] _Expr1 + _Expr2 = null when _Expr1 = Int, _Expr2 = Bool
[add12] _Expr1 + _Expr2 = null when _Expr1 = Int, _Expr2 = Str
[add13] _Expr1 + _Expr2 = null when _Expr1 = Int, _Expr2 = null
[add14] _Expr1 + _Expr2 = null when _Expr1 = Str, _Expr2 = Bool
[add15] _Expr1 + _Expr2 = null when _Expr1 = Str, _Expr2 = Int
[add16] _Expr1 + _Expr2 = null when _Expr1 = Str, _Expr2 = Str
[add17] _Expr1 + _Expr2 = null when _Expr1 = Str, _Expr2 = null
[add18] _Expr1 + _Expr2 = null when _Expr1 = null, _Expr2 = _Int
[add19] _Expr1 + _Expr2 = null when _Expr1 = null, _Expr2 = Str
[add20] _Expr1 + _Expr2 = null when _Expr1 = null, _Expr2 = null
[add21] _Expr1 + _Expr2 = null when _Expr1 = null, _Expr2 = null

[sub1] _Expr1 - _Expr2 = int(_Int1 - _Int2) when _Expr1 = Int, _Expr2 = Int

[sub2] _Expr1 - _Expr2 = real(_Real1 - _Real2) when _Expr1 = Real, _Expr2 = Real

[sub3] _Expr1 - _Expr2 = real(itor(_Int) - _Real) when _Expr1 = Int, _Expr2 = Real

[sub4] _Expr1 - _Expr2 = real(_Real - itor(_Int)) when _Expr1 = Real, _Expr2 = Int

[nul1] _Expr1 * _Expr2 = int(_Int1 * _Int2) when _Expr1 = Int, _Expr2 = Int

[nul2] _Expr1 * _Expr2 = real(_Real1 * _Real2) when _Expr1 = Real, _Expr2 = Real

[nul3] _Expr1 * _Expr2 = real(itor(_Int) * _Real) when _Expr1 = Int, _Expr2 = Real

[nul4] _Expr1 * _Expr2 = real(_Real * itor(_Int)) when _Expr1 = Real, _Expr2 = Int

[nul5] _Expr1 * _Expr2 = str(_Int * _Str) when _Expr1 = Int, _Expr2 = Str

[div1] _Expr1 / _Expr2 = int(_Int1 / _Int2) when _Expr1 = Int, _Expr2 = Int

[div2] _Expr1 / _Expr2 = real(_Real1 / _Real2) when _Expr1 = Real, _Expr2 = Real

[div3] _Expr1 / _Expr2 = real(itor(_Int) / _Real) when _Expr1 = Int, _Expr2 = Real

[div4] _Expr1 / _Expr2 = real(_Real / itor(_Int)) when _Expr1 = Real, _Expr2 = Int

[eq1] (_Expr1 = _Expr1 = bool(_Bool1 = _Bool2) when _Expr1 = Bool, _Expr2 = Bool
[eq2] (_Expr1 = _Expr1 = bool(_Bool1 = _Bool2) when _Expr1 = Bool, _Expr2 = Bool
[eq3] (_Expr1 = _Expr1 = null when _Expr1 = Bool, _Expr2 = Real
[eq4] (_Expr1 = _Expr1 = null when _Expr1 = Int, _Expr2 = Real
[eq5] (_Expr1 = _Expr1 = bool(itor(_Int) = _Real) when _Expr1 = Int, _Expr2 = Real
[eq6] (_Expr1 = _Expr1 = null when _Expr1 = Int, _Expr2 = Real
[eq7] (_Expr1 = _Expr1 = bool(_Real = itor(_Int)) when _Expr1 = Real, _Expr2 = Int
[eq8] (_Expr1 = _Expr1 = bool(_Real = itor(_Int)) when _Expr1 = Real, _Expr2 = Int
[eq9] (_Expr1 = _Expr1 = null when _Expr1 = Str, _Expr2 = Real
[eq10] (_Expr1 = _Expr1 = null when _Expr1 = Str, _Expr2 = Str
[eq11] (_Expr1 = _Expr1 = bool(_Str1 = _Str2) when _Expr1 = Str, _Expr2 = Str
[eq12] (_Expr1 = _Expr1 = null when _Expr1 = Str, _Expr2 = Str
[eq13] (_Expr1 = _Expr1 = null when _Expr1 = null, _Expr2 = null

[neq1] _Expr1 <> _Expr2 = (!(_Expr1 = _Expr2)

[gt1] (_Expr1 > _Expr2) = bool(_Int1 > _Int2) when _Expr1 = Int, _Expr2 = Int
[gt2] (_Expr1 > _Expr2) = bool(_Real1 > _Real2) when _Expr1 = Real, _Expr2 = Real
[gt3] (_Expr1 > _Expr2) = bool(itor(_Int) > _Real)
when _Expr1 = _Int, _Expr2 = _Real
[g4] (_Expr1 > _Expr2) = bool(_Real > itor(_Int))
when _Expr1 = _Real, _Expr2 = _Int
[g5] (_Expr1 > _Expr2) = bool(_Str1 > _Str2)
when _Expr1 = _Str1, _Expr2 = _Str2
[g6] (_Expr > null) = null
[g7] (null > _Expr) = null
[st1] _Expr1 < _Expr2 = (_Expr2 > _Expr1)
[gl1] _Expr1 := _Expr2 = ( _Expr1 > _Expr2) | (_Expr1 = _Expr2)
[ae1] _Expr1 <= _Expr2 = ( _Expr1 < _Expr2) | (_Expr1 = _Expr2)
[and1] _Expr1 & _Expr2 = bool(_Bool1 & _Bool2)
    when _Expr1 = _Bool1, _Expr2 = _Bool2
[and2] _Expr & null = bool(false) when _Expr = bool(false)
[and3] _Expr & null = null when _Expr != bool(false)
[and4] null & _Expr = bool(false) when _Expr = bool(false)
[and5] null & _Expr = null when _Expr != bool(false)
[and6] null & null = null
[or1] _Expr1 | _Expr2 = bool(_Bool1 | _Bool2)
    when _Expr1 = _Bool1, _Expr2 = _Bool2
[or2] _Expr | null = bool(true) when _Expr = bool(true)
[or3] _Expr | null = null when _Expr != bool(true)
[or4] null | _Expr = bool(true) when _Expr = bool(true)
[or5] null | _Expr = null when _Expr != bool(true)
[or6] null | null = null
[itor1] itor(_Expr) = real(itor(_Int)) when
    _Expr = _Int
[btos1] btos(_Expr) = str(btos(_Bool)) when
    _Expr = _Bool
[itos1] itos(_Expr) = str(itos(_Int)) when
    _Expr = _Int
[rtos1] rtos(_Expr) = str(rtos(_Real)) when
    _Expr = _Real

end Expressions
module Tuples

imports

sorts

VAL_TUP
VAR_TUP
EXPR_TUP
INST
INST_TUP

context-free syntax

"(" VAL ".")"* "-> VAL_TUP
"(" VAR ".")"* "-> VAR_TUP
"(" EXPFR ".")"* "-> EXPFR_TUP
VAR ':" VAL "-> INST
"(" INST ".")"* "-> INST_TUP

VAL_TUP "=" "-> Bool
inst "(" VAL_TUP "," VAL_TUP ")" "-> INST_TUP
bind "(" INST_TUP "," EXPFR ")" "-> EXPFR
bind "(" INST_TUP "," EXPFR_TUP ")" "-> EXPFR_TUP
val "(" EXPFR_TUP ")" "-> VAL_TUP

variables

"_VaL_TuP[0-9]"* "-> VAL_TUP
"_VaL_TuP[0-9]*" "-> (VAL_TUP "," ");
"_VaR_TuP[0-9]"* "-> VAL_TUP
"_ExPR_TuP[0-9]"* "-> EXPFR_TUP
"_InST[0-9]"* "-> INST
"_InST[0-9]*" "-> INST_TUP

hiddens

context-free syntax

eq "(" VAL_TUP "," VAL_TUP ")" "-> Bool

equations:

[eq1] (ValTup1 = ValTup2) = true
  when eq(ValTup1, ValTup2) = true
[eq2] (ValTup1 = ValTup2) = false
  when eq(ValTup1, ValTup2) = false

[eq3] eq(< >, < >) = true
[eq4] eq(< Val>, < >) = false
[eq5] eq(< >, < Val>) = false
[eq6] eq(< Bool1, Val1>, < Bool2, Val2>) =
    (Bool1 = Bool2) & eq(< Val1>, < Val2>)
[eq7] eq(< Int1, Val1>, < Int2, Val2>) =
    (Int1 = Int2) & eq(< Val1>, < Val2>)
[eq8] eq(< Real1, Val1>, < Real2, Val2>) =
    (Real1 = Real2) & eq(< Val1>, < Val2>)
[eq9] eq(< Str1, Val1>, < Str2, Val2>) =
    (Str1 = Str2) & eq(< Val1>, < Val2>)
[eq10] eq(< null, Val1>, < null, Val2>) =
    (Val1 = Val2)

[inst1] inst(< >, < >) = < >
[inst2] inst(< Var, _Var>, < Val, _Val>) = < Var : Val, _Inst>
  when inst(< Var>, < Val>) = < _Inst>
[inst3] inst(< Var>, < Val>) = < _Var>
[inst4] val(_Expr, _Expr) = < Val, _Val>
  when val(_Expr) = Val, val(_Expr) = _Val

[bind0] bind(< >, _Var) = _Var
[bind1] bind(< Var : _Bool, _Inst>, _Var) = _Bool
[bind2] bind(< Var, _Int, _Inst>, _Var) = _Int
[bind3] bind(< Var : Real, _Inst>, _Var) = _Real
[bind4] bind(< Var : _Str, _Inst>, _Var) = _Str
[bind5] bind(< Var : null, _Inst>, _Var) = _null
[bind6] bind(< Var : _Val, _Inst>, _Var') =
    bind(< _Inst>, _Var')
  when _Var' = _Var

[const1] bind(< InstTup, _Bool>) = _Bool
[const2] bind(InstTyp, Int) = Int
[const3] bind(InstTyp, Real) = Real
[const4] bind(InstTyp, Str) = Str
[const5] bind(InstTyp, null) = null

[not1] bind(InstTyp, !_Expr) = ! bind(InstTyp, _Expr)

[plus1] bind(InstTyp, +_Expr) = + bind(InstTyp, _Expr)

[min1] bind(InstTyp, -_Expr) = - bind(InstTyp, _Expr)

[add1] bind(InstTyp, _Expr1 + _Expr2) =
    bind(InstTyp, _Expr1) + bind(InstTyp, _Expr2)

[sub1] bind(InstTyp, _Expr1 - _Expr2) =
    bind(InstTyp, _Expr1) - bind(InstTyp, _Expr2)

[null1] bind(InstTyp, _Expr1 * _Expr2) =
    bind(InstTyp, _Expr1) * bind(InstTyp, _Expr2)

[div1] bind(InstTyp, _Expr1 / _Expr2) =
    bind(InstTyp, _Expr1) / bind(InstTyp, _Expr2)

[eql1] bind(InstTyp, _Expr1 = _Expr2) =
    bind(InstTyp, _Expr1) = bind(InstTyp, _Expr2)

[nqel1] bind(InstTyp, _Expr1 <> _Expr2) =
    bind(InstTyp, _Expr1) <> bind(InstTyp, _Expr2)

'gt1' bind(InstTyp, _Expr1 > _Expr2) =
    bind(InstTyp, _Expr1) > bind(InstTyp, _Expr2)

'lt1' bind(InstTyp, _Expr1 < _Expr2) =
    bind(InstTyp, _Expr1) < bind(InstTyp, _Expr2)

'[ge1] bind(InstTyp, _Expr1 >= _Expr2) =
    bind(InstTyp, _Expr1) >= bind(InstTyp, _Expr2)

'[le1] bind(InstTyp, _Expr1 <= _Expr2) =
    bind(InstTyp, _Expr1) <= bind(InstTyp, _Expr2)

[and1] bind(InstTyp, _Expr1 & _Expr2) =
    bind(InstTyp, _Expr1) & bind(InstTyp, _Expr2)

[or1] bind(InstTyp, _Expr1 | _Expr2) =
    bind(InstTyp, _Expr1) | bind(InstTyp, _Expr2)

[itor1] bind(InstTyp, itor(_Expr)) =
    itor(bind(InstTyp, _Expr))

[btos1] bind(InstTyp, btos(_Expr)) =
    btos(bind(InstTyp, _Expr))

[itos1] bind(InstTyp, itos(_Expr)) =
    itos(bind(InstTyp, _Expr))

[rtos1] bind(InstTyp, rtos(_Expr)) =
    rtos(bind(InstTyp, _Expr))

[tup1] bind(InstTyp, <>_Expr) = _Expr

[tup2] bind(InstTyp, <>_Expr, _Expr) =
    bind(InstTyp, _Expr, _Expr)
    when bind(InstTyp, _Expr) = _Expr

end Tuples
module Relations

imports

Layout
Booleans
Values
Expressions
Tuples

exports

sorts

REL

context-free syntax

VAR
"\{ (VAL_TUP \",\" )\}" -> REL
"!" REL -> REL
REL \"\&\" REL -> REL \{ left \}
REL \"\&\" REL -> REL \{ left \}
REL \"\#\" REL -> REL \{ left \}
\"( (REL \")\)\" -> REL \{ bracket \}

VAL_TUP \"8\" REL -> BOOL
\"\#\" REL -> INT

for all VAR_TUP \"3\" REL \";\" EXPR -> BOOL
these_is VAR_TUP \"3\" REL \";\" EXPR -> BOOL

sum \"( (REL \")\)\" -> EXPR
inf \"( (REL \")\)\" -> EXPR
sup \"( (REL \")\)\" -> EXPR

EXPR_TUP \"8\" REL -> EXPR

\"( (EXPR_TUP \"<\" VAR_TUP \"8\" REL \")\" \" EXPR \")\" -> REL
\"( (EXPR_TUP \"<\" VAR_TUP \"8\" REL \")\" -> REL
\"( (VAR_TUP \"8\" REL \")\" \" EXPR \")\" -> REL
\"( (VAR_TUP \"8\" REL \")\" -> REL

variables

\"_REL\"\{0-9\}*_ -> REL

hiddens

context-free syntax

rbind \"( (INST_TUP \",\" REL \")\" -> REL
scope-cover \"( (INST_TUP \",\" VAR_TUP \")\" -> INST_TUP
doesn't-occur \"( (VAR \",\" VAR_TUP \")\" -> BOOL

variables

\"_vt\"\{0-9\}*_ -> VAL_TUP
\"_vt\"\{0-9\}*_ -> (VAL_TUP \",\")
\"_vt\"\{0-9\}*_ -> (VAL_TUP \",\"

\%priorities: \# \> \+ ???

equations

\% [norm] [vt1, vt, vt*2, vt, vt*3] = [vt1, vt, vt*2, vt*3]

[unique1] (!) = (

[union] [\_vt1] + [\_vt2] = [\_vt1, \_vt2]

[diff1] [\_vt] = (
    when \_vt \& [\_vt2] = true
    when \_vt \& [\_vt2] = true

[inters1] [\_vt] * [\_vt] = 
    when \_vt \& [\_vt2] = true
    when \_vt \& [\_vt2] = true

\% case order of elements: (a, b) \# (x, y) = (ax, ay, bx, by)

[prod1] [\_vt] * [\_vt] = 
[prod2] [\_vt*1] * [\_vt] = 
[prod3] [\_vt1] * [\_vt2] =
[A1] for all _VarTup @ () : _Expr = bool(true)
[A2] for all _VarTup @ (_ValTup, _VarTup*) : _Expr =
  when bind(inst(_VarTup, _ValTup), _Expr) = bool(true)
[A3] for all _VarTup @ (_ValTup, _VarTup*) : _Expr = false
  when bind(inst(_VarTup, _ValTup), _Expr) = bool(false)

[E1] there is _VarTup @ () : _Expr = bool(false)
[E2] there is _VarTup @ (_ValTup, _VarTup*) : _Expr = bool(true)
  when bind(inst(_VarTup, _ValTup), _Expr) = bool(true)
[E3] there is _VarTup @ (_ValTup, _VarTup*) : _Expr =
  when bind(inst(_VarTup, _ValTup), _Expr) = bool(false)

[sim1] sum(() = null
[sim2] sum([<Int>,]) = Int
[sim3] sum([<Real>,]) = _Real
[sim4] sum([<Str>,]) = _Str
[sim5] sum([<Int>, <Int2>, _vt*]) = _Int1 + sum([<Int2>, _vt*])
[sim6] sum([<Real>, <Real2>, _vt*]) = _Real1 + sum([<Real2>, _vt*])
[sim7] sum([<Str>, <Str2>, _vt*]) = _Str1 + sum([<Str2>, _vt*])

[inf0] inf() = null
[inf1] inf([<Int>,]) = Int
[inf2] inf([<Real>,]) = _Real
[inf3] inf([<Str>,]) = _Str
[inf4] inf([<Int>, <Int2>, _vt*]) =
  inf(min([<Int>, <Int2>, _vt*])
[inf5] inf([<Real>, <Real2>, _vt*]) =
  inf(min([<Real>, <Real2>, _vt*])
[inf6] inf([<Str>, <Str2>, _vt*]) =
  inf([<Str>, _vt*]) when _Str2 > _Str1 = true
[inf7] inf([<Str>, <Str2>, _vt*]) =
  inf([<Str2>, _vt*]) when _Str1 > _Str2 = true

[sup0] sup() = null
[sup1] sup([<Int>,]) = Int
[sup2] sup([<Real>,]) = _Real
[sup3] sup([<Str>,]) = _Str
(sup4) sup([<Int>, <Int2>, _vt*]) =
  sup(max([<Int>, <Int2>, _vt*])
[sup5] sup([<Real>, <Real2>, _vt*]) =
  sup(max([<Real>, <Real2>, _vt*])
(sup6) sup([<Str>, <Str2>, _vt*]) =
  sup([<Str>, _vt*]) when _Str1 > _Str2 = true
[sup7] sup([<Str>, <Str2>, _vt*]) =
  sup([<Str2>, _vt*]) when _Str2 > _Str1 = true

end Relations
module DBSS

imports

Relations

exports

sorts

EXE
ENV
PROG
SCHEMA
DBASE
RDEF
RINST
STAT

context-free syntax

EXE  PROG  ->  EXE
SCHEMA  DBASE  ->  ENV
"schema"  RDEF  ->  SCHEMA
"database"  RINST  ->  DBASE
VAR  "--"  REL  ->  RDEF
VAR  "<<"  REL  ->  RDEF
VAR  "--"  REL  ->  RINST
"program"  STAT  "end"  ->  PROG
VAR  "<<"  REL  ";"  ->  STAT
"("  ExprTup  "<-"  "@"  REL  "|"  Expr  ")"  ->  REL
"("  "@"  REL  "|"  Expr  ")"  ->  REL
exec  "("  SCHEMA  ","  DBASE  ","  REL  ")"  ->  REL

variables

"Env"[0-9]+  ->  ENV
"Schema"[0-9]+  ->  SCHEMA
"DBase"[0-9]+  ->  DBASE
"RDef"[0-9]+  ->  RDEF
"RInst"[0-9]+  ->  RINST
"Stat"[0-9]+  ->  STAT
"Stat*"[0-9]+  ->  STAT*

hiddens

context-free syntax

pexec  "("  ENV  ","  STAT  ")"  ->  ENV
assign  "("  DBASE  ","  RINST  ")"  ->  DBASE
eval  "("  SCHEMA  ","  DBASE  ","  REL  ")"  ->  REL
lookup  "("  DBASE  ","  VAR  ")"  ->  REL
attrs  "("  SCHEMA  ","  REL  ")"  ->  VAR_TUP
attrs  "("  RDEF  ","  VAR  ")"  ->  VAR_TUP

equations

[ 0] exec(Schema, DBase, Rel) =
    ere(Schema, DBase, Rel)
 [ 1] _Env program _Stat _Stat* end =
    pexec(_Env, _Stat) program _Stat* end
 [ 2] pexec(Schema DBase, Var <= Rel) =
    ere(Schema, DBase, _Var = ere(Schema, DBase, _Rel))
 [ 3] assgn(database RInst1 Var = Rel1 RInst2, _Var = Rel2) =
    database RInst1 _Var = Rel2 RInst2
 [ 4] ere(Schema, DBase, _ValTup*) =
    [ _ValTup* ]
 [ 5] ere(Schema, DBase, !_Rel) =
    ! ere(Schema, DBase, _Rel)
 [ 6] ere(Schema, DBase, Rel1 + Rel2) =
    ere(Schema, DBase, Rel1) & ere(Schema, DBase, Rel2)
 [ 7] ere(Schema, DBase, Rel1 - Rel2) =
    ere(Schema, DBase, Rel1) & ! ere(Schema, DBase, Rel2)
 [ 8] ere(Schema, DBase, Rel1 * Rel2) =
    ere(Schema, DBase, Rel1) & ere(Schema, DBase, Rel2)
 [ 9] ere(Schema, DBase, Rel1 # Rel2) =
    ere(Schema, DBase, Rel1) & ! ere(Schema, DBase, Rel2)
 [10] ere(Schema, DBase, ExprTup <- VarTup & Rel | Expr) =
    ( ExprTup <- VarTup | ere(Schema, DBase, Rel) |
eval(Schema, Dbase, _Expr) =
  (VarTup $ @ Rel | _Expr) =
  (VarTup $ @rel(Schema, Dbase, _Rel) | eval(Schema, Dbase, _Expr))
eval(Schema, Dbase, Var) =
eval(Schema, Dbase, lookup(Dbase, Var))
lookup(database, Var) = Var
lookup(database Var = Rel RInst*, Var) = Rel
lookup(database Var = Rel RInst*, _Var) = lookup(database RInst*, _Var)
when _Var != _Var
  eval(Schema, Dbase, _Var) = var
  eval(Schema, Dbase, BoolConst) = BoolConst
  eval(Schema, Dbase, IntConst) = IntConst
  eval(Schema, Dbase, RealConst) = RealConst
  eval(Schema, Dbase, StrConst) = StrConst
  eval(Schema, Dbase, _Expr1 + _Expr2) =
    (eval(Schema, Dbase, _Expr1) + eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, _Expr1 - _Expr2) =
    (eval(Schema, Dbase, _Expr1) - eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, _Expr1 * _Expr2) =
    (eval(Schema, Dbase, _Expr1) * eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, _Expr1 / _Expr2) =
    (eval(Schema, Dbase, _Expr1) / eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, _Expr1 < _Expr2) =
    (eval(Schema, Dbase, _Expr1) < eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, _Expr1 > _Expr2) =
    (eval(Schema, Dbase, _Expr1) > eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, _Expr1 <= _Expr2) =
    (eval(Schema, Dbase, _Expr1) <= eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, _Expr1 >= _Expr2) =
    (eval(Schema, Dbase, _Expr1) >= eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, _Expr1 < _Expr2) =
    (eval(Schema, Dbase, _Expr1) < eval(Schema, Dbase, _Expr2))
  eval(Schema, Dbase, btos(_Expr)) =
    btos(eval(Schema, Dbase, _Expr))
  eval(Schema, Dbase, itos(_Expr)) =
    itos(eval(Schema, Dbase, _Expr))
  eval(Schema, Dbase, rtos(_Expr)) =
    rtos(eval(Schema, Dbase, _Expr))
  eval(Schema, Dbase, itor(_Expr)) =
    itor(eval(Schema, Dbase, _Expr))
  eval(Schema, Dbase, _Expr1 $ @ Rel) =
    _Expr1 $ @rel(Schema, Dbase, _Rel)
eval(Schema, Dbase, _Rel) =
  @rel(Schema, Dbase, _Rel)
eval(Schema, Dbase, sum(_Rel)) = sum(@rel(Schema, Dbase, _Rel))
eval(Schema, Dbase, inf(_Rel)) = inf(@rel(Schema, Dbase, _Rel))
eval(Schema, Dbase, sup(_Rel)) =
  sup(@rel(Schema, Dbase, _Rel))
eval(Schema, Dbase, _Expr1 <- _Rel | _Expr2) =
  eval(Schema, Dbase, _Expr2 @ attrf(Schema, _Rel) @ Rel)
eval(Schema, Dbase, _Expr1 <- attrf(Schema, _Rel) @ Rel | _Expr2) =
  attrf(Schema, _Rel) <- _Expr2
when attrf(Schema, _Rel) = <_Var>
  attrf(schema _Rel) = <_Var2>
attrf(schema _Rel) = <>
attrf(schema _Rel) = <Var1>
when attrf(schema _Rel) = <Var1>
  attrf(schema _Rel) = <Var2>
attrf(schema _Rel) = <Var3>
when attrf(schema _Rel) = <Var3>
  attrf(schema _Rel) = <Var4>
attrf(schema _Rel) = <Var5>
when attrf(schema _Rel) = <Var5>
  attrf(schema _Rel) = <Var6>
attrf(schema _Rel) = <Var7>
when attrf(schema _Rel) = <Var7>
  attrf(schema _Rel) = <Var8>
attrf(schema _Rel) = <Var9>
when attrf(schema _Rel) = <Var9>
  attrf(schema _Rel) = <Var10>
attrf(schema _Rel) = <Var11>
when attrf(schema _Rel) = <Var11>
  attrf(schema _Rel) = <Var12>
attrf(schema _Rel) = <Var13>
when attrf(schema _Rel) = <Var13>
  attrf(schema _Rel) = <Var14>
attrf(schema _Rel) = <Var15>
when attrf(schema _Rel) = <Var15>
end RDSS
module Lox_and_Names

exports

sorts
  Ident
  Unsigned
  Literal
  NumericLit
  ExactLit
  ApproxLit
  stringLit
  % StringElement
  TableName
  AuthorIdent
  TableIdent
  ColumnName
  CorrelationName
  ModuleName
  CursorName
  ProcedureName
  ParameterName
  EmbeddedVarName

lexical syntax
[ \n]             -> LAYOUT
" "              -> LAYOUT
[A-Za-z][A-20-9]*    -> Ident
(0-9)+            -> Unsigned
%% " / StringElement + " / " -> StringLit
%% " / \n  " StringElement
%% " / "  StringElement
%% " / [\n]+ " / " -> StringLit
plus " / Unsigned
" - " Unsigned
" - " ExactLit
" + " Unsigned
" + " ExactLit
" * " Unsigned
" * " ExactLit
" / " Unsigned
" / " ExactLit
 ExactLit "*" Unsigned -> ApproxLit
 ExactLit "*" Unsigned -> ApproxLit
 ExactLit "*" Unsigned -> ApproxLit
context-free syntax
  StringLit          -> Literal
  NumericLit         -> Literal
  ExactLit           -> NumericLit
  ApproxLit          -> NumericLit

% Names
AuthorIdent "." TableIdent    -> TableName
TableIdent              -> TableName
Ident                   -> AuthorIdent
Ident                   -> TableIdent
Ident                   -> ColumnName
Ident                   -> CorrelationName
Ident                   -> ModuleName
Ident                   -> CursorName
Ident                   -> ProcedureName
Ident                   -> ParameterName
Ident                   -> EmbeddedVarName  % see Annex A

variables
"_ident"[0-9]*    -> Ident
"_ident+"[0-9]*    -> Ident
"_unsigned"    -> Unsigned
"_stringLit"    -> StringLit
"_exactLit"    -> ExactLit
"_approxLit"    -> ApproxLit
"_literal"    -> Literal
"_tableName"    -> TableName
"_correlationName"    -> CorrelationName
"_parameterName"[0-9]*    -> ParameterName
"_embeddedVarName"[0-9]*    -> EmbeddedVarName

end Lox_and_Names
module ValueExpressions

imports
Lex_and_Names

exports
sorts
  ValueSpec
  ParameterSpec
  Variablespec
  TargetSpec
  indicatorParam
  IndicatorVar
  ColumnSpec
  Qualifier
  SetFuncSpec
  DistinctSetFunc
  AllSetFunc
  ValueExpr
  Term
  Factor
  Primary

context-free syntax

%% Value- and Target specification

ParameterSpec -> ValueSpec
Variablespec -> ValueSpec
Literal -> ValueSpec
"USER" -> ValueSpec
ParameterSpec -> TargetSpec
Variablespec -> TargetSpec
ParameterName IndicatorParam -> ParameterSpec
ParameterName -> ParameterSpec
"INDICATOR" ParameterName -> IndicatorParam
EmbeddedVarName IndicatorVar -> ValueSpec
EmbeddedVarName -> VariableSpec
"INDICATOR" EmbeddedVarName -> IndicatorVar

%% Column Spec
Qualifier "," ColumnName -> ColumnSpec
ColumnName -> ColumnSpec
TableName -> Qualifier
CorrelationName -> Qualifier

%% Set function specification

"COUNT" "+" Term -> SetFuncSpec
DistinctSetFunc -> SetFuncSpec
AllSetFunc -> SetFuncSpec
"AVG" "(" "DISTINCT" ColumnSpec ")" -> DistinctSetFunc
"MAX" "(" "DISTINCT" ColumnSpec ")" -> DistinctSetFunc
"MIN" "(" "DISTINCT" ColumnSpec ")" -> DistinctSetFunc
"SUM" "(" "DISTINCT" ColumnSpec ")" -> DistinctSetFunc
"AVG" "(" "ALL" ValueExpr ")" -> AllSetFunc
"MAX" "(" "ALL" ValueExpr ")" -> AllSetFunc
"MIN" "(" "ALL" ValueExpr ")" -> AllSetFunc
"SUM" "(" "ALL" ValueExpr ")" -> AllSetFunc
"AVG" "(" ValueExpr ")" -> AllSetFunc
"MAX" "(" ValueExpr ")" -> AllSetFunc
"MIN" "(" ValueExpr ")" -> AllSetFunc
"SUM" "(" ValueExpr ")" -> AllSetFunc

%% Value expressions

Term -> ValueExpr
ValueExpr "+" Term -> ValueExpr
ValueExpr "-" Term -> ValueExpr
Factor -> Term
Term "*" Factor -> Term
Term "/" Factora -> Term
"+" Primary -> Factor
"-" Primary -> Factor
Primary -> Factor
ValueSpec -> Primary
ColumnSpec -> Primary
SetFuncSpec -> Primary
"(" ValueExpr ")" -> Primary

variables

"ValueExpr" [0-9]* -> ValueExpr
"Term" [0-9]* -> Term
"Factor" [0-9]* -> Factor
"Primary" [0-9]* -> Primary
"ValueSpec" [0-9]* -> ValueSpec
"ColumnSpec" [0-9]* -> ColumnSpec
"Columnspec" -> (ColumnSpec ",")+
"ParameterSpec" -> Variablespec
"SetFuncSpec" -> SetFuncSpec

end ValueExpressions
module Predicates

imports
ValueExpressions

%% QuerySpecs

exports

sorts

Predicate
CompPredicate
CompOp
BetweenPredicate
InPredicate
InValueList
LikePredicate
Pattern
EscapeChar
NullPredicate
QuantifiedPredicate
Quantifier
All
Some
ExistsPredicate
SubQuery

%% See QuerySpecs for definition.

context-free syntax

%% Predicate
CompPredicate -> Predicate
BetweenPredicate -> Predicate
InPredicate -> Predicate
LikePredicate -> Predicate
NullPredicate -> Predicate
QuantifiedPredicate -> Predicate
ExistsPredicate -> Predicate

%% Comp predicate
ValueExpr CompOp ValueExpr -> CompPredicate
ValueExpr CompOp SubQuery -> CompPredicate
"=" -> CompOp
"<>" -> CompOp
"<" -> CompOp
">" -> CompOp
"<=" -> CompOp
"=>" -> CompOp

%% Between predicate
ValueExpr "NOT" "BETWEEN" ValueExpr "AND" ValueExpr -> BetweenPredicate
ValueExpr "BETWEEN" ValueExpr "AND" ValueExpr -> BetweenPredicate

%% In predicate
ValueExpr "NOT" "IS" SubQuery -> InPredicate
ValueExpr "NOT" "IS" InValueList -> InPredicate
ValueExpr "IS" SubQuery -> InPredicate
ValueExpr "IS" InValueList -> InPredicate
[ ValueSpec "," ] + -> InValueList

%% Like predicate
ColumnSpec "NOT" "LIKE" Pattern "ESCAPE" EscapeChar -> LikePredicate
ColumnSpec "NOT" "LIKE" Pattern -> LikePredicate
ColumnSpec "LIKE" Pattern "ESCAPE" EscapeChar -> LikePredicate
ColumnSpec "LIKE" Pattern -> LikePredicate
ValueSpec -> Pattern
ValueSpec -> EscapeChar

%% Null predicate
ColumnSpec "IS" "NOT" "NULL" -> NullPredicate
ColumnSpec "IS" "NULL" -> NullPredicate

%% Quantified predicate
ValueExpr CompOp Quantifier SubQuery -> QuantifiedPredicate
All -> Quantifier
Some -> Quantifier
"ALL" -> All
"SOME" -> Some
"ANY" -> Some

%% Exists predicate
"EXISTS" SubQuery -> ExistsPredicate

variables

"_Predicate" -> Predicate
"_CompOp" -> CompOp
"_InValueList" -> InValueList
"_ValueSpec" -> (ValueSpec "," ")+
"_Pattern" -> Pattern
"_EscapeChar" -> EscapeChar
"_Quantifier" -> Quantifier

end Predicates
module QuerySpecs
imports
LexAndNames
ValueExpressions
Predicates
exports
sorts
SearchCondition
BooleanTerm
BooleanFactor
BooleanPrimary
TableExpr
FromClause
TableReference
WhereClause
GroupByClause
HavingClause
%% SubQuery : declared in Predicates.
ResultSpec
QuerySpec
 SelectList
context-free syntax
%% search conditions
BooleanTerm
SearchCondition "OR" BooleanTerm
BooleanFactor
BooleanTerm "AND" BooleanFactor
"NOT" BooleanPrimary
BooleanPrimary
Predicate
(" SearchCondition ")
-> BooleanPrimary
%% Table expression
FromClause WhereClause GroupByClause HavingClause
-> TableExpr
FromClause WhereClause GroupByClause HavingClause -> TableExpr
FromClause WhereClause GroupByClause HavingClause
-> TableExpr
FromClause GroupByClause HavingClause
-> TableExpr
FromClause GroupByClause
-> TableExpr
FromClause HavingClause
-> TableExpr
FromClause
-> TableExpr
%% From clause
"FROM" [ TableReference "," ] +
-> FromClause
TableName CorrelationName
-> TableName
-> TableReference
%% Where clause
"WHERE" SearchCondition
-> WhereClause
%% Group by clause
"GROUP" "BY" [ Columnspec "," ] +
-> GroupByClause
%% Having clause
"HAVING" SearchCondition
-> HavingClause
%% Subquery
(" 'SELECT' "ALL" ResultSpec TableExpr ")
-> SubQuery
(" 'SELECT' "DISTINCT" ResultSpec TableExpr ")
-> SubQuery
(" 'SELECT' ResultSpec TableExpr ")
-> SubQuery
ValueExpr
-> ResultSpec
-> ResultSpec
%% Query specification
"SELECT" "ALL" SelectList TableExpr
-> QuerySpec
"SELECT" "DISTINCT" SelectList TableExpr
-> QuerySpec
"SELECT" SelectList TableExpr
-> QuerySpec
[ ValueExpr "," ] +
-> SelectList
-> SelectList
%% variables
"_QuerySpec"
-> QuerySpec
"_SubQuery"
-> SubQuery
"_ResultSpec"
-> ResultSpec
"_SelectList"
-> SelectList
"_ValueExpr"
-> (ValueExpr "," )+
"_TableExpr"
-> TableExpr
"_FromClause"
-> FromClause
"_WhereClause"
-> WhereClause
"_GroupByClause"
-> GroupByClause
"_HavingClause"
-> HavingClause
"_TableRefList"
-> (TableReference "," )+
"_SearchCondition"
-> SearchCondition
"_BooleanTerm"
-> BooleanTerm
"_BooleanFactor"
-> BooleanFactor
"_BooleanPrimary"
-> BooleanPrimary
"_ColumnspecList"
-> (Columnspec "," )+
"_TableRef"
-> TableReference
"_TableRef+
-> (TableReference "," )+
module DataTypes
imports
Lex_and_Names
exports

sorts
DataType
StringType
ExactType
ApproxType
Length
Precision
Scale

context-free syntax

%% DataTypes

StringType -> DataType

ExactType  -> DataType

ApproxType -> DataType

"CHARACTER" "(" Length ")" -> StringType

"CHARACTER" -> StringType

"CHAR" "(" Length ")" -> StringType

"CHAR" -> StringType

"NUMERIC" "(" Precision "," Scale ")" -> ExactType

"NUMERIC" "(" Precision ")" -> ExactType

"NUMERIC" -> ExactType

"DECIMAL" "(" Precision "," Scale ")" -> ExactType

"DECIMAL" "(" Precision ")" -> ExactType

"DECIMAL" -> ExactType

"DEC" "(" Precision "," Scale ")" -> ExactType

"DEC" "(" Precision ")" -> ExactType

"DEC" -> ExactType

"INTEGER" -> ExactType

"LENGTH" -> ExactType

"REAL" -> ExactType

"DOUBLE" "PRECISION" -> ApproxType

Unsigned  -> Length

Unsigned  -> Precision

Unsigned  -> Scale

variables

"_Length"[0-9]* -> Length

"_Precision"[0-9]* -> Precision

"_Scale"[0-9]* -> Scale

equations

[5-05-SR-1] CHAR = CHARACTER
[5-05-SR-1] CHAR(_Length) = CHARACTER(_Length)
[5-05-SR-1] DEC = DECIMAL
[5-05-SR-1] DEC(_Precision) = DECIMAL(_Precision)
[5-05-SR-1] DEC(_Precision, _Scale) = DECIMAL(_Precision, _Scale)
[5-05-SR-1] INT = INTEGER

[5-05-SR-3] CHARACTER = CHARACTER(1)
[5-05-SR-3] NUMERIC = NUMERIC(13)
[5-05-SR-3] DECIMAL = DECIMAL(13)
[5-05-SR-3] FLOAT = FLOAT(13)
[5-05-SR-3] NUMERIC(_Precision) = NUMERIC(_Precision, 0)
[5-05-SR-3] DECIMAL(_Precision) = DECIMAL(_Precision, 0)

end DataTypes
module Schemas

imports
Lex_and.Names
DataTypes
QuerySpecs

exports

sorts
Environment
Schema
SchemaAuthorClause
SchemaAuthorIdent
SchemaElement
Tab.eDef
Tab.eElement
ColumnDef
ColumnConstr
DefaultClause
TableConstrDef
UniqueConstrDef
UniqueSpec
UniqueColumnList
ReferentialConstrDef
ReferencesSpec
ReferencingColumns
ReferencedTableAndColumns
ReferenceColumnList
CheckConstrDef
ViewDef
ViewColumnList
PrivilegesDef
Privileges
Action
GrantColumnList
Grantee

class syntax

%% Environment
"CREATE" "ENVIRONMENT" Ident Schema* -> Environment

%% Schema
"CREATE" "SCHEMA" SchemaAuthorClause SchemaElement* -> Schema
"AUTHORIZATION" SchemaAuthorIdent -> SchemaAuthorClause
AuthorIdent
TableDef -> SchemaElement
ViewDef -> SchemaElement
PrivilegesDef -> SchemaElement

%% Table Def
"CREATE" "TABLE" TableName "("[TableElement"]","+" ")" -> TableDef

ColumnDef -> TableElement
TableConstrDef -> TableElement

%% Column Def
ColumnName DataType DefaultClause ColumnConstr* -> ColumnDef
ColumnName DataType ColumnConstr* -> ColumnDef
"NOT" "NULL" UniqueSpec -> ColumnConstr
"NOT" "NULL" -> columnConstr
ReferencesSpec
"CHECK" "("SearchCondition ")" -> ColumnConstr

%% Default Clause
"DEFAULT" Literal -> DefaultClause
"DEFAULT" "USER" -> DefaultClause
"DEFAULT" "NULL" -> DefaultClause

%% Table Constr Def
UniqueConstrDef -> TableConstrDef
ReferentialConstrDef -> TableConstrDef
CheckConstrDef -> TableConstrDef

%% Unique Constr Def
UniqueSpec "(" UniqueColumnList ")" -> UniqueConstrDef
"UNIQUE" -> UniqueSpec
"PRIMARY" "KEY" -> UniqueSpec
% Referential Constr Def

"FOREIGN "KEY "(" ReferencingColumns ")" ReferencesSpec -> ReferentialConstrDef
"REFERENCES" ReferencedTableAndColumns ReferencingColumns
TableName "(" ReferenceColumnList ")" -> ReferencedTableAndColumns
TableName [ ColumnName "," ] * -> ReferenceColumnList

% check Constr Def

"CHECK" "(" SearchCondition ")" -> CheckConstrDef
%

% View Def

"CREATE" "VIEW" TableName "(" ViewColumnList ")" -> ViewDef
"AS" QuerySpec "WITH" "CHECK" "OPTION" -> ViewDef
"CREATE" "VIEW" TableName -> ViewDef
"AS" QuerySpec "WITH" "CHECK" "OPTION" -> ViewDef
"CREATE" "VIEW" TableName -> ViewDef
"AS" QuerySpec -> ViewDef

{ ColumnName "," }* -> ViewColumnList
%

% Privilege Def

"GRANT" Privileges "ON" TableName
"TO" [ Grantee "," ]* "WITH" "GRANT" "OPTION" -> PrivilegeDef
"GRANT" Privileges "ON" TableName "TO" [ Grantee "," ]* -> PrivilegeDef

"ALL" "PRIVILEGES"-> Privileges
{ Action "," }* -> Privileges
"SELECT" -> Action
"INSERT" -> Action
"DELETE" -> Action
"UPDATE" "(" GrantColumnList ")" -> Action
"UPDATE" -> Action

{ ColumnName "," }* -> GrantColumnList

% PUBLIC
AuthID -> Grantee

variables
"Environment" -> Environment
"Schema" -> Schema
"SchemaElement" -> SchemaElement
"ColumnsElement"* -> SchemaElement
"TableElement" -> TableElement
"TableElement" "(" TableElement "," ")"* -> TableDef
"ViewDef" -> ViewDef
"PrivilegeDef" -> PrivilegeDef
"ColumnDef" -> ColumnDef
"TableConstrDef" -> TableConstrDef
"ViewColumnList" -> ViewColumnList
"QuerySpec" -> QuerySpec
"Privileges" -> Privileges
"GranteeList" [ Grantee "," ]* -> Grantee
"Grantee" -> Grantee
"Grantee" -> Grantee
"ColumnDef" "(" ColumnDef ")"* -> Column
"ColumnName" -> ColumnName
"ColumnSet" -> ColumnSet
"FunctionDef" -> FunctionDef
"StringType" -> StringType
"ExactType" -> ExactType
"ApproxType" -> ApproxType
"Literal" -> Literal
"UniqueConstrDef" -> UniqueConstrDef
"ReferentialConstrDef" -> ReferentialConstrDef
"CheckConstrDef" -> CheckConstrDef

end Schemas
module DataManipulation

imports
QuerySpec
DataTypes

exports

sorts
CloseStat
CommitStat
DeclareCursor
CursorSpec
QueryExpr
QueryTerm
OrderByClause
SortSpec
DeleteStatPositioned
DeleteStatSearchedList
FetchStat
FetchTargetList
InsertStat
InsertColumnList
InsertValueList
InsertValue
OpenStat
RollbackStat
SelectStat
SelectTargetList
UpdateStatPositioned
SetClausePositioned
ObjectColumnPositioned
UpdateStatSearchedList
SetClauseSearchedList
ObjectColumnSearchedList

class-free syntax

%% Data manipulation Language

%% Close Stat
"CLOSE" CursorName  \rightarrow\  CloseStat

%% Commit Stat
"COMMIT" "WORK"  \rightarrow\  CommitStat

%% Declare Cursor
"DECLARE" CursorName "CURSOR" "FOR" CursorSpec  \rightarrow\  DeclareCursor

QueryExpr OrderByClause  \rightarrow\  CursorSpec
QueryExpr  \rightarrow\  CursorSpec

QueryTerm  \rightarrow\  QueryExpr
QueryExpr "UNION" "ALL" QueryTerm  \rightarrow\  QueryExpr
QueryExpr "UNION" QueryTerm  \rightarrow\  QueryExpr

SortSpec  \rightarrow\  QueryTerm
(" QueryExpr ")  \rightarrow\  QueryTerm

"ORDER" "BY" \{SortSpec ","\}*  \rightarrow\  OrderByClause

Unsigned "ASC"  \rightarrow\  SortSpec
Unsigned "DESC"  \rightarrow\  SortSpec
Unsigned  \rightarrow\  SortSpec
ColumnSpec "ASC"  \rightarrow\  SortSpec
ColumnSpec "DESC"  \rightarrow\  SortSpec
ColumnSpec

%% Delete Stat Positioned
"DELETE" "FROM" TableName  \rightarrow\  DeleteStatPositioned

WHERE" "CURRENT" "OF" CursorName  \rightarrow\  DeleteStatPositioned

%% Delete Stat SearchedList
"DELETE" "FROM" TableName "WHERE" SearchCondition  \rightarrow\  DeleteStatSearchedList

DELETE" "FROM" TableName  \rightarrow\  DeleteStatSearchedList

%% Fetch Stat
"FETCH" CursorName "INTO" FetchTargetList  \rightarrow\  FetchStat

\{TargetSpec ","\}*,*  \rightarrow\  FetchTargetList
\{ParameterSpec ","\}*,*  \rightarrow\  FetchTargetList
### Insert Stat

"INSERT" "INTO" TableName "(" InsertColumnList ")"
"VALUES" "(" InsertValueList ")" -> InsertStat
"INSERT" "INTO" TableName "(" InsertColumnList ")" QuerySpec; -> InsertStat
"INSERT" "INTO" TableName "(" InsertColumnList ")" -> InsertStat
"VALUES" "(" InsertValueList ")" -> InsertStat
"INSERT" "INTO" TableName QuerySpec -> InsertStat

(ColumnName ",",)* -> InsertColumnList
(InsertValue ",",)* -> InsertValueList
ValueSpec -> InsertValue
"NULL" -> InsertValue

### Open Stat

"OPEN" CursorName -> OpenStat

### Rollback Stat

"ROLLBACK" "WORK" -> RollbackStat

### Select Stat

"SELECT" "ALL" SelectList "INTO" SelectTargetList TableExpr -> SelectStat
"SELECT" "DISTINCT" SelectList "INTO" SelectTargetList TableExpr -> SelectStat
"SELECT" SelectList "INTO" SelectTargetList TableExpr -> SelectStat

(TargetSpec ",",)* -> SelectTargetList
(ParameterSpec ",",)* -> SelectTargetList

### Update Stat Positioned

"UPDATE" TableName "SET" (SetClausePositioned ",",)*
"WHERE" "CURRENT" "OF" CursorName -> UpdateStatPositioned
ObjectColumnPositioned "=" ValueExpr -> SetClausePositioned
ObjectColumnPositioned "=" "NULL" -> SetClausePositioned
ColumnName -> ObjectColumnPositioned

### Update Stat Searched

"UPDATE" TableName "SET" (SetClauseSearched ",",)* "WHERE" SearchCondition -> UpdateStatSearched
"UPDATE" TableName "SET" (SetClauseSearched ",",)* -> UpdateStatSearched
ObjectColumnSearched "=" ValueExpr -> SetClauseSearched
ObjectColumnSearched "=" "NULL" -> SetClauseSearched
ColumnName -> ObjectColumnSearched

end DataManipulation
module Modules

imports
Lex_and_Names
DataTypes
DataManipulation

exports

sorts
ModuleNameClause
LanguageClause
ModuleAuthorClause
ModuleAuthorIdent
ModuleNameClause
Procedure
ParamDeclaration
SQLCODEFParam
SQLStat

context-free syntax

%% Module Language

%% module
ModuleNameClause
LanguageClause
ModuleAuthorClause
DeclaraCursor
Procedure
"LANGUAGE" "COBOL"
"LANGUAGE" "FORTRAN"
"LANGUAGE" "PASCAL"
"LANGUAGE" "PLI"
"AUTHORIZATION" ModuleAuthorIdent
AuthorIdent

%% Module Name Clause

"MODULE" ModuleName
"MODULE"

%% Procedure

"PROCEDURE" ProcedureName ParamDeclaration+ ";" SQLStat ";" -> Procedure

ParameterName DataTypes
SQLCODEFParam
"SQLCODE"

CloseStat
CommitStat
DeleteStatPositioned
DeleteStatSearched
FetchStat
InsertStat
OpenStat
RollbackStat
SelectStat
UpdateStatPositioned
UpdateStatSearched

end Modules

module SQL

imports
Schemas
QuerySpecs
Moduler

end SQL
module Errors
imports
Layout
exports
sorts
ERROR

lexical syntax
"OK"      -> ERROR
"ERROR!"  -> ERROR

context-free syntax
ERROR "AND" ERROR                     -> ERROR (right)
ERROR "AND" "THEN" ERROR              -> ERROR (right)
"(" ERROR ")"                         -> ERROR

variables
"_Error"[0-9]*                        -> ERROR

equations
[err1] _OK AND _Error = _Error
[err2] _Error AND _OK = _Error
[err3] _OK AND _THEN _Error = _Error
[err4] _Error1 AND _THEN _Error2 = _Error1
[err5] (_Error) = _Error

end Errors
module SchemaDescr

imports

Layout
Booleans
Integers
Schemas

exports

sorts
EnvDescr % Environment
SListDescr % Schema
EDescr % Schema
TVListDescr % (Table<View>Def)
TVDescr % (Table<View>Def
CollistDescr % ColumnDef
ColDescr % ColumnDef
DTDescr % DataType
TypeDescr % Type
DefDescr % DefaultCause
CConListDescr % ColumnConstdef
TConListDescr % TableConstDef
TCollDescr % TableConstDef
IdectList % ViewColumnDescr, ...
PLDescr % PrivilegeDef
PDescr % PrivilegeDef
GListDescr % Grantee

context-free syntax

"...

"< [Ident ",", SLListDescr "]" -> EnvDescr
"[ EDescr ]" -> EListDescr
"< [Ident ",", TVListDescr ",", PListDescr ]" -> PListDescr

%% Table Description
"< [Ident ",", Ident ",", CollistDescr ",", TConListDescr ]" -> TVDescr
"[ ColDescr ]" -> TVListDescr
"< [Ident ",", DTDescr ",", DefDescr ",", CConListDescr ",", BOCL ]" -> TVDescr

		"String" -> TVDescr
		"Exact" -> TVDescr
		"Approx" -> TVDescr
		"Notype" -> TVDescr
		"Literal" -> TVDescr
		"USER" -> TVDescr
		"NULL"
"[ [ColumnConstdef ] ]" -> CConListDescr
"[ [TableConstdef ] ]" -> TConListDescr
"< UniqueConstDef ",", ReferentialConstDef ",", CheckConstDef ]" -> TConListDescr

%% View Description
"< [Ident ",", Ident ",", Collist ",", QuerySpec ",", BOCL ]" -> TVDescr

"[ [Ident ] ]" -> TVDescr

%% Privilege Description
"[ [PDescr ] ]" -> TVDescr
"< [Privileges ",", Ident ",", Ident ",", GLDescr ",", BOCL ]" -> TVDescr

"[ [Grantees ] ]" -> TVDescr


env-descr "[ Environment ]" -> EnvDescr

%%% auxiliary functions for using the description:

lookup-table "[ TableName ",", EnvDescr ]" -> TVListDescr
lookup-table "[ TableName ",", EDescr ]" -> TVListDescr
lookup-table "[ TableName ",", TVListDescr ]" -> TVListDescr
lookup-table "[ TableName ",", TVDescr ]" -> TVListDescr
eq-table-name "[ TableName ",", TableName ]" -> BOOL
eq-table-id "[ TableIdent ",", TableIdent ]" -> BOOL
corrsname "[ CorrelationName ",", TVDescr ]" -> TVDescr
combine-tables "[ TVListDescr ",", TVListDescr ]" -> TVListDescr
append-table "[ TVListDescr ",", TVDescr ]" -> TVListDescr
set-group-column "[ ColumnSpec ",", TVListDescr ]" -> TVListDescr
set-group-column "[ ColumnSpec ",", TVDescr ]" -> TVDescr
set-group-column "[ ColumnSpec ",", CollistDescr ]" -> CollistDescr
set-group-column "[ ColumnSpec ",", CollDescr ]" -> CollDescr
lookup-column "[ ColumnSpec ",", TVListDescr ]" -> CollistDescr
lookup-column "(" ColumnSpec "," TVDescr ")"  -> ColListDescr
lookup-column "(" ColumnSpec "," ColListDescr ")"  -> ColListDescr
lookup-column "(" ColumnSpec "," ColDescr ")"  -> ColListDescr

variables
"_EvDescr"[0-9]*  -> EvDescr
"_SListDescr"  -> SListDescr
"_SDescr"[0-9]*  -> SDescr
"_TDescr"[0-9]*  -> TDescr
"_TVListDescr"[0-9]*  -> TVListDescr
"_TVDescr+"  -> TVDescr
"_TVDescr"[0-9]*  -> TVDescr
"_TVDescr"[0-9]*  -> TVDescr
"_ColListDescr"  -> ColListDescr
"_ColDescr"  -> ColDescr
"_ColDescr[0-9]*"  -> ColDescr
"_DTypeDescr[0-9]*"  -> DtypeDescr
"_TypDescr"[0-9]*  -> TypDescr
"_DatDescr"  -> DatDescr
"_CConListDescr"  -> CConListDescr
"_CConListDescr"  -> CConListDescr
"_CConDescr"  -> CConDescr
"_CConDescr"  -> CConDescr

"_IdenList"  -> IdenList
"_PListDescr[0-9]*"  -> PListDescr
"_FDescr"  -> FDescr
"_FDescr"  -> FDescr

hiddens
context-free syntax

sl - descr "(" Schema ")"  -> SListDescr
s - descr "(" Schema ")"  -> SDescr
tvl - descr "(" Ident ","," SchemaElement "")"  -> TVListDescr
tv - descr "(" Ident ","," SchemaElement "")"  -> TVDescr
coll - descr "(" (TableElement ") invoices" )"  -> ColListDescr
coll - descr "(" (ColumnDef ") first" )"  -> ColDescr
dt - descr "(" (DataType ") number" )"  -> DtypeDescr
type "(" (DataType ") number" )"  -> TypDescr
length "(" (DataType ") number" )"  -> Unsigned
scale "(" (DataType ") number" )"  -> Unsigned
default "(" (DefaultClause ") NULL" )"  -> DefaultDescr
coll - descr "(" ColumnConstraint ") invoices" )"  -> CConListDescr
tconl - descr "(" (TableElement ") invoices" )"  -> TConDescr
tconl - descr "(" (TableConstraintDef ") " invoices" )"  -> TConDescr
tv - descr "(" (ViewColumnList ") " invoices" )"  -> IdentList
pl - descr "(" Ident ","," SchemaElement "")"  -> PListDescr
p - descr "(" Ident ","," PrivilegeDef "")"  -> FDescr
gl - descr "(" (Grantee ") " invoices" )"  -> SLDescr

unsignedInt "(" (Unsigned ") " invoices" )"  -> NAT

equations:

[envl] env - descr(CREATE ENVIRONMENT _Ident _Schema)* =
(Ident, sl - descr(_Schema))*

[sl1] sl - descr() = []
[sl2] sl - descr(_Schema _Schema)* =
[sl3] when sl - descr(_Schema) = (_SDescr)

[sl1] s - descr(CREATE SCHEMA AUTHORIZATION _Ident _SchemaElement*) =
(Ident, tvl - descr(_Ident, _SchemaElement)), pl - descr(_Ident, _SchemaElement)*

[tv1] tvl - descr(_Ident, ) = []
[tv2] tvl - descr(_Ident, _TableDef _SchemaElement*) =
[tv3] tvl - descr(_Ident, _TableDef _TVDescr*) when tvl - descr(_Ident, _SchemaElement) = (_TVDescr)
[tv4] tvl - descr(_Ident, _ViewDef _SchemaElement*) =
[tv5] tvl - descr(_Ident, _ViewDef _TVDescr*) when tvl - descr(_Ident, _SchemaElement) = (_TVDescr)
[tv6] tvl - descr(_Ident, _PrivilegeDef _SchemaElement*) =
[tv7] tvl - descr(_Ident, _PrivilegeDef _TVDescr*)

[tv1] tvl - descr(_Ident, CREATE TABLE _Ident1 _Ident2 _TableElement*) =
(Ident, _Ident1, _Ident2, coll - descr(_TableElement), tconl - descr(_TableElement))
[tv2] tvl - descr(_Ident, CREATE TABLE _Ident _TableElement*) =
(Ident, _Ident, _Ident2, coll - descr(_TableElement), tconl - descr(_TableElement))
[tv3] tvl - descr(_Ident, CREATE VIEW _Ident1 _Ident2 _ViewColumnList)
AS _QuerySpec WITH CHECK OPTION) -
<Ident1, Ident2, idl-descr(ViewColumnList), _QuerySpec, true>
[tv4] tv-descr(_Ident, CREATE VIEW _Ident1._Ident2 ( _ViewColumnList )
AS _QuerySpec) =
<Ident1, Ident2, idl-descr(ViewColumnList), _QuerySpec, false>
[tv5] tv-descr(_Ident, CREATE VIEW _Ident1._Ident2
AS _QuerySpec WITH CHECK OPTION) =
<Ident1, Ident2, idl-descr(ViewColumnList), _QuerySpec, true>
[tv6] tv-descr(_Ident, CREATE VIEW _Ident1._Ident2
AS _QuerySpec) =
<Ident1, Ident2, idl-descr(ViewColumnList), _QuerySpec, false>
[tv7] tv-descr(_Ident, CREATE VIEW _Ident1._Ident2 ( _ViewColumnList )
AS _QuerySpec WITH CHECK OPTION) =
<Ident1, Ident2, idl-descr(ViewColumnList), _QuerySpec, true>
[tv8] tv-descr(_Ident, CREATE VIEW _Ident1._Ident2
AS _QuerySpec) =
<Ident1, Ident2, idl-descr(ViewColumnList), _QuerySpec, false>
[tv9] tv-descr(_Ident, CREATE VIEW _Ident1
AS _QuerySpec WITH CHECK OPTION) =
<Ident1, Ident2, idl-descr(ViewColumnList), _QuerySpec, true>
[tv10] tv-descr(_Ident, CREATE VIEW _Ident2
AS _QuerySpec) =
<Ident1, Ident2, idl-descr(ViewColumnList), _QuerySpec, false>

[co11] co1-descr(_ColumnDef) =
[co12] co1-descr(_TableConstDef) =
[[]]
[co13] co1-descr(_ColumnDef, _TableElement*) =
[co1-descr(_ColumnDef) _ColDescr*]
when co1-descr(_TableElement*) = [ColDescr]*
[co14] co1-descr(_TableConstDef, _TableElement*) =
[ColDescr*]
when co1-descr(_TableElement*) = [ColDescr*]
[co15] co1-descr(_Ident _DataType DefaultClause _ColumnConst*) =
<Ident, dt-descr(_DataType), default(DefaultClause), co1-descr(_ColumnConst*), false>
[co16] co1-descr(_Ident _DataType _ColumnConst*) =
<Ident, dt-descr(_DataType), NULL, co1-descr(_ColumnConst*), false>

%d default "NULL" may be the wrong choice (see 6.3.
[dt1] dt-descr(_DataType) =
	$type(_DataType), unsigned2int(length(_DataType)), unsigned2int(scalar(_DataType))>

%%  type() and scalar(): see below.
[def1] default(DEFAULT_LITERAL) = _Literal
[def2] default(DEFAULT USER) = _USER
[def3] default(DEFAULT NULL) = NULL
[co11] co1-descr(_ColumnConst*) = [ColumnConst]*
[tcon11] tcon1-descr(_ColumnDef) =
[]
[tcon12] tcon1-descr(_TableConstDef) =
[tcon1-descr(_TableConstDef)]
[tcon13] tcon1-descr(_ColumnDef, _TableElement*) =
[TColDescr*]
when tcon1-descr(_TableElement*) = [TColDescr]*
[tcon14] tcon1-descr(_TableConstDef, _TableElement*) =
[tcon1-descr(_TableConstDef) TColDescr*]
when tcon1-descr(_TableElement*) = [TColDescr]*
[tcon1] tcon1-descr(_UniqueConstDef _ReferentialConstDef _CheckConstDef) =
<UniqueConstDef, _ReferentialConstDef, _CheckConstDef>

[id11] idl-descr(_Ident) = [Ident]
[id12] idl-descr(_Ident, _ColumnName*) =
[Ident _Ident]
when idl-descr(_ColumnName*) = [Ident]
[pl1] pl-descr(_Ident, _Ident) = []
[pl12] pl-descr(_Ident, _TableDef SchemaElement*) =
[pl-descr(_Ident, _SchemaElement*)]
[pl13] pl-descr(_Ident, _ViewDef SchemaElement*) =
[pl-descr(_Ident, _SchemaElement*)]
[pl14] pl-descr(_Ident, _PrivilegeDef SchemaElement*) =
[p-descr(_Ident, _PrivilegeDef _PDescr*)]
when pl-descr(_Ident, _SchemaElement*) = [PDescr]*
[p1] p-descr(_Ident, GRANT Privileges ON _Ident1._Ident2 TO _GranteeList WITH GRANT OPTION) =
<Privileges, _Ident1._Ident2, gl-descr(_GranteeList), true>
[p2] p-descr(_Ident, GRANT Privileges ON _Ident1._Ident2 TO _GranteeList) =
<Privileges, _Ident1._Ident2, gl-descr(_GranteeList), false>
[p3] p-descr(_Ident, GRANT Privileges ON _Ident1._Ident2 TO _GranteeList WITH GRANT OPTION) =
<Privileges, _Ident1._Ident2, gl-descr(_GranteeList), true>
[p4] p-descr(_Ident, GRANT Privileges ON _Ident1._Ident2 TO _GranteeList) =
<Privileges, _Ident1._Ident2, gl-descr(_GranteeList), false>

[gl1] gl-descr() = []
[gl2] gl-descr(_Grantee, _GranteeList) =
[Grantee, Grantee*]
when gl-descr(_GranteeList) = [Grantee]*

% type(); length() and scale();
% Als gevolg van de equations van module DataTypes zijn niet alle gevallen-
% onderscheidingen noodzakelijk voor de functies type, length en scale
% hierop. Daarom wordt hier op de juistste wijze gerepresenteerd:
% length(CHAR) -> length(CHARACTER) -> length(CHARACTER(1)) -> 1

{ttype1 : type(StringType) = String
{ttype2 : type(ExactType) = Exact
{ttype3 : type(ApproxType) = Approx

{len1 : length(LENGTH) = Length
{len2 : length(NUMERIC(Precision, Scale)) = Precision
{len3 : length(DECIMAL(Precision, Scale)) = Precision
{len4 : length(INTEGER) = 13
{len5 : length(SMALLINT) = 7
{len6 : length(REAL) = Precision
{len7 : length(REAL) = 13
{len8 : length(DOUBLE PRECISION) = 26

{scale1 : scale(LENGTH) = 0
{scale2 : scale(NUMERIC(Precision, Scale)) = Scale
{scale3 : scale(DECIMAL(Precision, Scale)) = Scale
{scale4 : scale(INTEGER) = 0
{scale5 : scale(SMALLINT) = 0
{scale6 : scale(REAL) = 147
{scale7 : scale(REAL) = 147
{scale8 : scale(DOUBLE PRECISION) = 147

{uInt2 : unsignedInt2(_unsigned_Value) = int_const(_Char)

% Auxiliary functions for using the descriptions:

{look1 : lookup-table(Table, _Id, []) = []
{look2 : lookup-table(Table, _Id, [SDescr SDescr]) = [SDescr[0] TVDescr]*
when lookup-table(Table, _Id, TVDescr) = [TVDescr[0] TVDescr]*
lookup-table(Table, _Id, [SDescr SDescr]) = [TVDescr[0] TVDescr]*

{look3 : lookup-table(Table, _Id, TVListDescr) = lookup-table(Table, _Id, TVListDescr)

{look4 : lookup-table(Table, [], []) = []
{look5 : lookup-table(Table, [TVDescr TVDescr]) = [TVDescr[0] TVDescr]*
when lookup-table(Table, TVDescr) = [TVDescr[0] TVDescr]*
lookup-table(Table, TVDescr) = [TVDescr[0] TVDescr]*

{look7 : lookup-table(Table, _Id1, _Id2, CollListDescr, _ConListDescr) = [{_Id1, _Id2, CollListDescr, _ConListDescr}
when eq-table-name(Table, _Id1, _Id2) = true
lookup-table(Table, _Id1, _Id2, CollListDescr, _ConListDescr) = []
when eq-table-name(Table, _Id1, _Id2) = false

{look8 : lookup-table(Table, _Id1, _Id2, _IdList, _QuerySpec, _Bool) = [{_Id1, _Id2, _IdList, _QuerySpec, _Bool}
when eq-table-name(Table, _Id1, _Id2) = true
lookup-table(Table, _Id1, _Id2, _IdList, _QuerySpec, _Bool) = []
when eq-table-name(Table, _Id1, _Id2) = false

{eq1 : eq-table-name(_Id1, _Id2, _Id3, _Id4) = eq-table-id(_Id1, _Id3) & eq-table-id(_Id2, _Id4)
{eq2 : eq-table-name(_Id1, _Id2, _Id4) = eq-table-id(_Id2, _Id4) & eq-table-id(_Id2, _Id4)
{eq3 : eq-table-name(_Id1, _Id2, _Id4) = eq-table-id(_Id2, _Id4) & eq-table-id(_Id2, _Id4)
{eq4 : eq-table-name(_Id2, _Id4) = eq-table-id(_Id2, _Id4) & eq-table-id(_Id2, _Id4)

{eq5 : _Id1 = _Id2 => eq-table-id(_Id1, _Id2) = true
{eq6 : _Id1 = _Id2 => eq-table-id(_Id1, _Id2) = false

{look9 : lookup-column(ColumnSpec, []) = []
{look10 : lookup-column(ColumnSpec, [TVDescr TVDescr]) = [ColDescr[0] ColDescr[2]]
when lookup-column(ColumnSpec, TVDescr) = [ColDescr[0] ColDescr[2]]
lookup-column(ColumnSpec, TVDescr) = [ColDescr[0] ColDescr[2]]

{look11 : lookup-column(Table, ColumnName, _Id1, _Id2, CollListDescr, _ConListDescr) = lookup-column(Table, ColumnName, CollListDescr)
when eq-table-name(Table, _Id1, _Id2) = true
lookup-column(Table, ColumnName, _Id1, _Id2, CollListDescr, _ConListDescr) = []
when eq-table-name(Table, _Id1, _Id2) = false
lookup-column(Table, ColumnName, _Id1, _Id2, CollListDescr) = lookup-column(Table, ColumnName, CollListDescr)
lookup-column([ColumnName, []] = []
[looko7] lookup-column([ColumnName, [ColDescr ColDescr*]] =
[ColDescr*1, ColDescr*2]
    when lookup-column([ColumnName, ColDescr] = [ColDescr*1],
        lookup-column([ColumnName, [ColDescr*1]] = [ColDescr*2])
[looko8] lookup-column([Ident, <Ident>, DTDescr, Literal, _ConListDescr, _Bool]) =
    [<Ident>, DTDescr, Literal, _ConListDescr, _Bool>
    when _Ident = _Ident'
[looko9] lookup-column([Ident, <Ident>, DTDescr, Literal, _ConListDescr, _Bool]) =
    []
    when _Ident = _Ident'

++ combine-tables(old, new) :
++ combine-tables([A B C], [D E A B]) = [D E A B]
[comb2] combine-tables([[TVDescr TVDescr*1], [TVDescr*2]]) =
    combine-tables([[TVDescr*1], append-table([[TVDescr*2], TVDescr]])

[appt1] append-table([[TVDescr*1], <Ident1, _Ident2, _ConListDescr, _ConListDescr>] =
    [[TVDescr*1]
    when lookup-table([Ident1, _Ident2, tvDescr*1]) != []
[appt2] append-table([[TVDescr*1], <Ident1, _Ident2, _ConListDescr, _ConListDescr>] =
    [[TVDescr*1] <Ident1, _Ident2, _ConListDescr, _ConListDescr>
    when lookup-table([Ident1, _Ident2, tvDescr*1]) != []
[appt3] append-table([[TVDescr*2], <Ident1, _Ident2, _ConList, QuerySpec, _Bool>] =
    [[TVDescr*2] <Ident1, _Ident2, _ConList, QuerySpec, _Bool>
    when lookup-table([Ident1, _Ident2, tvDescr*2]) != []
[appt4] append-table([[TVDescr*2], <Ident1, _Ident2, _ConList, QuerySpec, _Bool>] =
    [[TVDescr*2] <Ident1, _Ident2, _ConList, QuerySpec, _Bool>
    when lookup-table([Ident1, _Ident2, tvDescr*2]) != []
[sao01] set-group-column([ColumnSpec, []] = []
[sao02] set-group-column([ColumnSpec, TVDescr TVDescr*1]) =
    [TVDescr TVDescr*1]
    when set-group-column([ColumnSpec, TVDescr*1]) = _TVDescr'
[sao03] set-group-column([TableSpec, TableName, [Ident1, _Ident2, _ConListDescr, _ConListDescr]] =
    <Ident1, _Ident2, set-group-column([TableSpec, _ConListDescr, _ConListDescr]
    when eq-table-name([TableName, _Ident1, _Ident2]) = true
[sao04] set-group-column([TableSpec, TableName, [Ident1, _Ident2, _ConListDescr, _ConListDescr]] =
    <Ident1, _Ident2, _ConListDescr, _ConListDescr>
    when eq-table-name([TableName, _Ident1, _Ident2]) = false
[sao05] set-group-column([ColumnSpec, _Ident1, _Ident2, _ConListDescr, _ConListDescr]) =
    
[sao06] set-group-column([Ident, []] = []
[sao07] set-group-column([Ident, ColDescr ColDescr*1]) =
    [ColDescr*1 ColDescr*2]
    when set-group-column([Ident, ColDescr] = _ColDescr'
[sao08] set-group-column([Ident, <Ident>, DTDescr, Literal, _ConListDescr, _Bool]) =
    [<Ident>, DTDescr, Literal, _ConListDescr, _Bool>
    when _Ident = _Ident'
[sao09] set-group-column([Ident, <Ident>, DTDescr, Literal, _ConListDescr, _Bool]) =
    [<Ident>, DTDescr, Literal, _ConListDescr, _Bool>
    when _Ident = _Ident'
[cor10] correname([Ident, _Ident2, _ConListDescr, _ConListDescr]) =
    <Ident1, _Ident2, _ConListDescr, _ConListDescr>
[cor11] correname([Ident, _Ident2, _ConList, QuerySpec, _Bool]) =
    <Ident1, _Ident2, _ConList, QuerySpec, _Bool>
end SchemaDescr
module SchemaCheck

imports
Errors SchemaDescr TC_Queryped

exports

context-free syntax

to ("EnvDescr") -> ERROR
expand-env ("EnvDescr", "SListDescr") -> EnvDescr
expand-sch ("EnvDescr", "SDescr") -> SDescr
expand-tv ("EnvDescr", "TVDescr") -> TVDescr

hiddens

context-free syntax

s1-check ("EnvDescr", "SListDescr") -> ERROR
s-check ("EnvDescr", "SDescr") -> ERROR
tv-check ("EnvDescr", "Ident", "TListDescr") -> ERROR
tv-check ("EnvDescr", "TVDescr") -> ERROR
coll-check ("CollListDescr") -> ERROR
vocoll-check ("IdentList", "CollListDescr") -> ERROR
should-provide-viweolumnlist ("IdentList", "CollListDescr") -> ERROR
all-cda-must-be-different ("IdentList") -> ERROR
right-number-of-viweolumn ("IdentList", "CollListDescr") -> ERROR

"Schema Id" Ident "should be unique!" -> ERROR
"Authorization Id" Ident "should be unique!" -> ERROR
"Table Name" Ident "should be unique!" -> ERROR
"Column Name" Ident "should be unique!" -> ERROR
"Should provide viweolumnlist!" -> ERROR
"All viewolumn must be different!" -> ERROR
"Too many viewolumn specified!" -> ERROR
"Not enough viewolumn specified!" -> ERROR
scheck ("SDescr") -> Ident
s-occurs ("Ident", "SListDescr") -> BOOL
author-id ("TVDescr") -> Ident
table-id ("TVDescr") -> Ident
column-id ("CollListDescr") -> BOOL
c-occurs ("Ident", "CollListDescr") -> BOOL
i-occurs ("Ident", "IdentList") -> BOOL
rename ("IdentList", "CollListDescr") -> Ident

envi to (Ident, SListDescr) =
  s1-check (Ident, SListDescr, _SListDescr)

[s1]  s1-check (EnvDescr, []) = OK
[s12] s1-check (EnvDescr, [SDescr1 SDescr1 SDescr2]) = % volgorde afhankelijk impl!
  Schema Id schema-id SDescr should be Unique!
  s1-check (EnvDescr, [SDescr1 SDescr2]) = true
[s13] s1-check (EnvDescr, [SDescr1 SDescr2]) =
  s-check (EnvDescr, _SDescr) AND s-check (EnvDescr, [SDescr1 SDescr2])
  when s-occurs (schema-id (SDescr), [SDescr1 SDescr2]) = false

[s1d1] schema-id (Ident, _TListDescr, _SListDescr) = Ident

[soc1] s-occurs (Ident, []) = false
[soc2] s-occurs (Ident, [SDescr SDescr]) = true
when Ident = schema-id (SDescr)
[soc3] s-occurs (Ident, [SDescr SDescr]) =
  s-occurs (Ident, [SDescr])
when Ident = schema-id (SDescr)

[1]  s-check (EnvDescr, _Ident, TListDescr, _SListDescr) =
  tvi-check (EnvDescr, _Ident, _TVDescr)

[tv1] tvi-check (EnvDescr, _Ident, []) = OK
[tv12] tvi-check (EnvDescr, _Ident, [TVDescr1 TVDescr2 TVDescr2]) =
  Authorization Id author-id (TVDescr) should be Ident!
  when author-id (TVDescr) = Ident
[tv13] tvi-check (EnvDescr, _Ident, [TVDescr1 TVDescr2 TVDescr2]) =
  Table Name table-id (TVDescr) should be unique!
  when t-occurs (table-id (TVDescr), [TVDescr1 TVDescr2]) = true
[tv14] tvi-check (EnvDescr, _Ident, [TVDescr1 TVDescr2 TVDescr2]) =
  when author-id (TVDescr) = Ident,
  t-occurs (table-id (TVDescr), [TVDescr1 TVDescr2]) = false

[auth1] author-id (Ident1, Ident2, CollListDescr, _ConListDescr) = Ident1
[auth2] author-id (Ident1, Ident2, CollListDescr, _ConListDescr) = Ident1

taid (Ident1, Ident2, CollListDescr, _ConListDescr) = Ident2
[tid2] table-id (Ident1, Ident2, IdentList, QuerySpec, Bool) = Ident2

toc12] t-occurs (Ident1, []) = false
[toc2] t-occurs (Ident1, [TVDescr TVDescr]) = true
when Ident = table-id (TVDescr)
toc3] t-occurs (Ident1, [TVDescr TVDescr]) =
  t-occurs (Ident1, [TVDescr])
when Ident = table-id (TVDescr)

[tv1] tvi-check (EnvDescr, _Ident1, _Ident2, CollListDescr, _ConListDescr) =
  coll-check (CollListDescr)
[tv2] tvi-check (EnvDescr, _Ident1, _Ident2, IdentList, QuerySpec, Bool) =
  _Error
module C2_ValueExpr
imports Errors SchemaDescr ValueExpressions
exports sorts
  VE_EXPR % _EDP = Error Description Pair
context-free syntax
  "<ERROR "","_ DTDesc ">" -> VE_EXPR
  "Column" ColumnSpec "undefined!" -> ERROR
  "Column" ColumnSpec "ambiguous!" -> ERROR
  "Type of operand(s) of StringLit "may not be string!" -> ERROR
  "Type of func-arg "ValueExpr "may not be string!" -> ERROR
  "Parameters not implemented!" -> ERROR
  "Variables not implemented!" -> ERROR
  we-check "(" TVListDescr "," ValueExpr ")" -> VE_EXPR
hiddens
context-free syntax
  c-check "(" TVListDescr "," ColumnSpec ")" -> VE_EXPR
  exp-check "(" StringLit "," VE_EXPR ",&" VE_EXPR ")" -> VE_EXPR
  exp-type "+" TypDescr "," TypDescr ")" -> TypDescr
  exp-prec "+" "," INT "," INT ")" -> INT
  exp-scale "(" StringLit ",&" INT "," INT ")" -> INT
  length "(" StringLit ")" -> INT
  length "(" Unsigned ")" -> INT
  precision "(" ExactLit ")" -> INT
  scale "(" ExactLit ")" -> INT
  precision "(" ApproxLit ")" -> INT

equations
  [v01] we-check(TVListDescr, ValueExpr + _Term) =
    exp-check('+', we-check(TVListDescr, ValueExpr), we-check(TVListDescr, _Term))
  [v02] we-check(TVListDescr, ValueExpr - _Term) =
    exp-check('-', we-check(TVListDescr, ValueExpr), we-check(TVListDescr, _Term))
  [v03] we-check(TVListDescr, _Term + Factor) =
    exp-check('+', we-check(TVListDescr, _Term), we-check(TVListDescr, Factor))
  [v04] we-check(TVListDescr, _Term / Factor) =
    exp-check('/', we-check(TVListDescr, _Term), we-check(TVListDescr, Factor))
  [v05] we-check(TVListDescr, + Primary) =
    exp-check('+', we-check(TVListDescr, Primary), we-check(TVListDescr, Primary))
  [v06] we-check(TVListDescr, - Primary) =
    exp-check('-', we-check(TVListDescr, Primary), we-check(TVListDescr, Primary))
  [v07] we-check(TVListDescr, ParameterName INDICATOR ParameterName') =
    <Parameters not implemented!; Notype,0,0>>
  [v08] we-check(TVListDescr, EmbeddedVarName INDICATOR EmbeddedVarName') =
    <Variables not implemented!, Notype,0,0>>
  [v09] we-check(TVListDescr, EmbeddedVarName) =
    <Variables not implemented!, Notype,0,0>>
  [v10] we-check(TVListDescr, StringLit) =
    <OK, <String, length(StringLit), 0>>
  [v11] we-check(TVListDescr, ExactLit) =
    <OK, <Exact, precision(ExactLit), scale(ExactLit)>>
  [v12] we-check(TVListDescr, ApproxLit) =
    <OK, <Approx, precision(ApproxLit), 0>>
  [v13] we-check(TVListDescr, USER) =
    <OK, <String, 18, 0>> % length is impl.defined; 18 = max length of ident
  [v14] we-check(TVListDescr, ColumnSpec) =
    c-check(TVListDescr, ColumnSpec)
  [v15] we-check(TVListDescr, _ValueExpr) =
    we-check(TVListDescr, ValueExpr)
  [v16] we-check(TVListDescr, COUNT(*) ) =
    <OK, <Exact, 1113, 0>> % precision impl-def
  [v17] we-check(TVListDescr, COUNT(DISTINCT ColumnSpec)) =
    <Error, Notype,0,0>
  when c-check(TVListDescr, ColumnSpec) = <Error, DTDescr>,
    Error = OK
  [v18] we-check(TVListDescr, MAX(DISTINCT ColumnSpec)) =
    c-check(TVListDescr, ColumnSpec)
  [v19] we-check(TVListDescr, MIN(DISTINCT ColumnSpec)) =
    c-check(TVListDescr, ColumnSpec)
  [v20] we-check(TVListDescr, MAX(ALL ValueExpr)) =
    we-check(TVListDescr, ValueExpr)
  [v21] we-check(TVListDescr, MIN(ALL ValueExpr)) =
    we-check(TVListDescr, ValueExpr)
  [v22] we-check(TVListDescr, MAX(ALL ValueExpr)) =
    we-check(TVListDescr, ValueExpr)
  [v23] we-check(TVListDescr, MIN(ALL ValueExpr)) =
    we-check(TVListDescr, ValueExpr)
  [v24] we-check(TVListDescr, AVG(DISTINCT ColumnSpec)) =
    we-check(TVListDescr, AVG(ColumnSpec))
  \footnote{a ColumnSpec IS a ValueExpr!}
  [v25] we-check(TVListDescr, AVG(ALL ValueExpr)) =
va-check (_TVListDescr, AVG(_ValueExpr))
when va-check (_TVListDescr, _ValueExpr) =<OK, _TypeDescr, _Int2>,
    _ValueExpr =<OK, _TypeDescr, _Int2>,
    _TypeDescr = String
va-check (_TVListDescr, AVG(_ValueExpr)) =<Type of fmp-arg _ValueExpr may be not string, <Notype, 0, 0>>
when va-check (_TVListDescr, _ValueExpr) =<OK, _TypeDescr, _Int2>,
    _TypeDescr =<String, _Int1, _Int2>,
    _ValueExpr =<Notype, 0, 0>
va-check (_TVListDescr, SUM(DISTINCT _ColumnSpec)) =
va-check (_TVListDescr, SUM(_ColumnSpec))
  % a columnspec is a ValueExpr!
va-check (_TVListDescr, SUM(ALL _ValueExpr)) =
va-check (_TVListDescr, SUM(_ValueExpr))
va-check (_TVListDescr, SUM(_ValueExpr))

[01] c-check (_TVListDescr, _ColumnSpec) =<OK, _TypeDescr, _ColumnSpec>
when lookup-column (_ColumnSpec, _TVListDescr) =<ColDescr, _ColDescr =<Ident, _TypeDescr, _Literal, _CONListDescr, _Bool>>
[02] c-check (_TVListDescr, _ColumnSpec) =<Column _ColumnSpec undefined, <Notype, 0, 0>>
when lookup-column (_ColumnSpec, _TVListDescr) =[]
[03] c-check (_TVListDescr, _ColumnSpec) =<Column _ColumnSpec ambiguous, <Notype, 0, 0>>
when lookup-column (_ColumnSpec, _TVListDescr) =<ColDescr _ColDescr _ColDescr>

[exp1] exp-check (StringLit, _Error1, _TypeDescr1, _Error2, _TypeDescr2) =<Error1 AND _Error2, _Notype, 0, 0>
when _Error1 AND _Error2 = OK
[exp2] exp-check (StringLit, _Error1, _TypeDescr1, _Error2, _TypeDescr2) =<Type of operand(s) of _StringLit may not be string (as notype), <Notype, 0, 0>>
when _Error1 = OK, _Error2 = OK,
    _TypeDescr1 =<String, _Int1, _Int2>
[exp3] exp-check (StringLit, _Error1, _TypeDescr1, _Error2, _TypeDescr2) =<Type of operand(s) of _StringLit may not be string (as notype), <Notype, 0, 0>>
when _Error1 = OK, _Error2 = OK,
    _TypeDescr1 =<String, _Int1, _Int2>
[exp4] exp-check (StringLit, _Error1, _TypeDescr1, _Error2, _TypeDescr2) =<OK, <exp-type (_TypeDescr1, _TypeDescr2),
    exp-pw(_Int1, _Int2),
    exp-scale(_StringLit, _Int1, _Int2)>>
when _Error1 = OK, _Error2 = OK,
    _TypeDescr1 =<TypeDescr, _Int1, _Int1, _Int1>, _TypeDescr2 =<String, _Int1, _Int1, _Int1>, _TypeDescr2 =<String
[exp1] exp-type (Exact, Exact) = Exact
[exp2] exp-type (Exact, Approx) = Approx
[exp3] exp-type (Approx, Exact) = Approx
[exp4] exp-type (Approx, Approx) = Approx

[exp1] exp-pw (_Int1, _Int2) = max (_Int2, _Int1) % implanmentor-defined
[exp2] exp-scale ("", _Int1, _Int2) = max (_Int1, _Int2) % eig alleen voor exact
[exp3] exp-scale ("", _Int1, _Int2) = max (_Int1, _Int2) % eig alleen voor exact
[exp4] exp-scale ("", _Int1, _Int2) = max (_Int1, _Int2) % eig alleen voor exact

[len1] length(stringLit("" _Char "")) = 1
[len2] length(stringLit("" _Char _Char+ ""))
[len3] length(stringLit("" _Char _Char+ ""))
[len4] length(stringLit("" _Char _Char+ ""))
[len5] length(stringLit("" _Char _Char+ ""))
[len6] length(stringLit("" _Char _Char+ ""))
[len7] length(stringLit("" _Char _Char+ ""))
[len8] length(stringLit("" _Char _Char+ ""))
[len9] length(stringLit("" _Char _Char+ ""))

[pw1] precision(exactlit (_"" _Char+""))
[pw2] precision(exactlit (_"" _Char+""))
[pw3] precision(exactlit (_"" _Char+""))
[pw4] precision(exactlit (_"" _Char+""))
[pw5] precision(exactlit (_"" _Char+""))
[pw6] precision(exactlit (_"" _Char+""))
[pw7] precision(exactlit (_"" _Char+""))
[pw8] precision(exactlit (_"" _Char+""))
[pw9] precision(exactlit (_"" _Char+""))
[pw10] precision(exactlit (_"" _Char+""))

[scale1] scale(exactlit (_"" _Char+""))
[scale2] scale(exactlit (_"" _Char+""))
[scale3] scale(exactlit (_"" _Char+""))
[scale4] scale(exactlit (_"" _Char+""))
[scale5] scale(exactlit (_"" _Char+""))
[scale6] scale(exactlit (_"" _Char+""))
[scale7] scale(exactlit (_"" _Char+""))
[scale8] scale(exactlit (_"" _Char+""))
[scale9] scale(exactlit (_"" _Char+""))
[scale10] scale(exactlit (_"" _Char+""))
module CC_QueryPred_syn

imports

Errors
Predicates
SchemaDescr
Queryspecs
TC_ValueExpr

exports

sorts

QUERY_EDP
SELL_EDP
TEXPED
FROM_EDP
WHERE_EDP
GROUP_EDP
HAVING_EDP
SEARCH_EDP
TABLE_EDP

context-free syntax

"< 'ERROR ',' , ',' Ident ',' , ',' CollListDescr ',' >"  -> QUERY_EDP
"< 'ERROR ',' , ',' CollListDescr ',' >"  -> SELL_EDP
"< 'ERROR ',' , ',' TVListDescr ',' >"  -> TEXPED
"< 'ERROR ',' , ',' TVListDescr ',' >"  -> FROM_EDP
"< 'ERROR ',' , ',' WHERE ',' >"  -> WHERE_EDP
"< 'ERROR ',' , ',' GROUP ',' >"  -> GROUP_EDP
"< 'ERROR ',' , ',' HAVING ',' >"  -> HAVING_EDP
"< 'ERROR ',' , ',' TABLE ',' >"  -> TABLE_EDP

"Table Ref" TableName "undefined;"  -> ERROR
"Table Ref" TableName "ambiguous;"  -> ERROR

query-check "(" EnmDescr "," QuerySpec ")"  -> QUERY_EDP
sell-check "(" EnmDescr "," TVListDescr "," SelectList ")"  -> SELL_EDP
texp-check "(" EnmDescr "," TVListDescr "," TableExpr ")"  -> TEXPED
from-check "(" EnmDescr "," FromClause ")"  -> FROM_EDP
where-check "(" EnmDescr "," TVListDescr "," WhereClause ")"  -> WHERE_EDP
group-check "(" TVListDescr "," GroupByClause ")"  -> GROUP_EDP
having-check "(" EnmDescr "," TVListDescr "," HavingClause "," ")"  -> HAVING_EDP
search-check "(" EnmDescr "," TVListDescr "," SearchCondition ")"  -> SEARCH_EDP
table-check "(" EnmDescr "," TableReference ")"  -> TABLE_EDP
pred-check "(" EnmDescr "," TVListDescr "," Predicate ")"  -> ERROR
subq-check "(" EnmDescr "," TVListDescr "," SubQuery ")"  -> QUERY_EDP
subsel-check "(" EnmDescr "," TVListDescr "," SubQuery ")"  -> VEXPED
column-check "(" CollListDescr ")"  -> VEXPED

comparable "(" VEXPED "," VEXPED ")"  -> ERROR
comparable "(" DTDescr "," DTDescr ")"  -> BOOL

variables

"QUERY_EDP"  -> QUERY_EDP
"SELL_EDP"  -> SELL_EDP
"TEXPED"  -> TEXPED
"FROM_EDP"  -> FROM_EDP

end CC_QueryPred_syn
module TC_Pred_sem

imports

TC_QueryPred_syn

exports

context-free syntax

"SubQuery must result in a single column!"  -> ERROR
"Types are not comparable!"  -> ERROR
"Type of ValueExp must be a string!"  -> ERROR
"Value" ValueExp must be a single char!"  -> ERROR

hiddens

context-free syntax

must-be-string "(" TVListDescr "," ValueExp ")"  -> ERROR
must-be-length "(" TVListDescr "," ValueExp ")"  -> ERROR

equations

[pred1] pred-check(EvDescri, TVListDescr, ValueExpr CompOp ValueExpr2) =
comparable(ve-check(TVListDescr, ValueExpr1), ve-check(TVListDescr, ValueExpr2))

[pred2] pred-check(EvDescri, TVListDescr, ValueExpr CompOp Subquery) =
comparable(ve-check(TVListDescr, ValueExpr), subjval-check(EvDescri, TVListDescr, Subquery))

[pred3] pred-check(EvDescri, TVListDescr, ValueExpr NOT BETWEEN ValueExpr2 AND ValueExpr3) =
comparable(ve-check(TVListDescr, ValueExpr), ve-check(TVListDescr, ValueExpr2)) AND
comparable(ve-check(TVListDescr, ValueExpr), ve-check(TVListDescr, ValueExpr3))

[pred4] pred-check(EvDescri, TVListDescr, ValueExpr BETWEEN ValueExpr2 AND ValueExpr3) =
comparable(ve-check(TVListDescr, ValueExpr), ve-check(TVListDescr, ValueExpr2)) AND
comparable(ve-check(TVListDescr, ValueExpr), ve-check(TVListDescr, ValueExpr3))

[pred5] pred-check(EvDescri, TVListDescr, ValueExpr IN Subquery) =
comparable(ve-check(TVListDescr, ValueExpr), subjval-check(EvDescri, TVListDescr, Subquery))

[pred6] pred-check(EvDescri, TVListDescr, ValueExpr IN ValueSpec) =
comparable(ve-check(TVListDescr, ValueExpr), ve-check(TVListDescr, ValueSpec))

[pred7] pred-check(EvDescri, TVListDescr, ValueExpr IN ValueSpec, ValueSpec2) =
comparable(ve-check(TVListDescr, ValueExpr), ve-check(TVListDescr, ValueSpec)) AND
THEN

[pred8] pred-check(EvDescri, TVListDescr, ValueExpr NOT IN Subquery) =
pred(EvDescri, TVListDescr, ValueExpr IN Subquery)

[pred9] pred-check(EvDescri, TVListDescr, ValueExpr NOT IN TVValueList) =
pred(EvDescri, TVListDescr, ValueExpr IN TVValueList)

[pred10] pred-check(EvDescri, TVListDescr, ColumnSpec LIKE ValueSpec ESCAPE ValueSpec2) =
must-be-string(TVListDescr, ColumnSpec) AND
must-be-string(TVListDescr, ValueSpec) AND

[pred11] pred-check(EvDescri, TVListDescr, ColumnSpec LIKE ValueSpec) =
must-be-string(TVListDescr, ColumnSpec) AND
must-be-string(TVListDescr, ValueSpec)

[pred12] pred-check(EvDescri, TVListDescr, ColumnSpec NOT LIKE ValueSpec ESCAPE ValueSpec2) =
pred(EvDescri, TVListDescr, ColumnSpec LIKE ValueSpec ESCAPE ValueSpec2)

[pred13] pred-check(EvDescri, TVListDescr, ColumnSpec NOT LIKE ValueSpec) =
pred(EvDescri, TVListDescr, ColumnSpec LIKE ValueSpec)

[pred14] pred(EvDescri, TVListDescr, ColumnSpec IS NULL) =
Error
when v-check(TVListDescr, ColumnSpec) = (Error, DTDescr)

[pred15] pred(EvDescri, TVListDescr, ColumnSpec IS NOT NULL) =
pred(EvDescri, TVListDescr, ColumnSpec IS NOT NULL)

[pred16] pred(EvDescri, TVListDescr, ValueExpr CompOp Quantifier Subquery) =
comparable(ve-check(TVListDescr, ValueExpr), subjval-check(EvDescri, TVListDescr, Subquery))

[pred17] pred(EvDescri, TVListDescr, EXISTS Subquery) =
Error
when subjval-check(EvDescri, TVListDescr, Subquery) = (<Error, _Ident, _CollListDescr>

[sqrel1] subjval-check(EvDescri, TVListDescr, _Subquery) =
<Error, _Notype, 0, 0>
when subjval-check(EvDescri, TVListDescr, _Subquery) = (<Error, _Ident, _CollListDescr>,
_Error) = OK

[sqrel2] subjval-check(EvDescri, TVListDescr, _Subquery) =
<Error, DTDescr>
when subjval-check(EvDescri, TVListDescr, _Subquery) = (<_Notype, 0, 0>)

[colm1] column-check[] = (Error, _Notype, 0, 0)  44 something went wrong!

[colm2] column-check(!!CollDescr CollDescr) =
<Error, DTDescr>
when column-check(!!CollDescr, CollDescr) =
column-check(CollDescr) = (Error, DTDescr)

[colm3] column-check(<_Notype, DTDescr, _DefDescr, _ConListDescr, _Bool>) =
<OK, DTDescr>

[compl1] comparable(Error1, DTDescr) =
Error1 AND Error2
when Error1 AND Error2 = OK

[compl2] comparable(OK, DTDescr), OK =
Types are not comparable!
when comparable(DTDescr, DTDescr) = true 44 cave: niet test op false!

[compl3] comparable(OK, DTDescr), OK =
Types are not comparable!
when comparable(DTDescr, DTDescr) = true 44 cave: niet test op false!
OK
when comparable(_DTDescr1, _DTDescr2) = true

[comp4] comparable(<String, _Int1, _Int2>, <String, _Int1*, _Int2*>) = true
[comp5] comparable(<Exact, _Int1, _Int2>, <Exact, _Int1*, _Int2*>) = true
[comp6] comparable(<Approx, _Int1, _Int2>, <Approx, _Int1*, _Int2*>) = true
[comp7] comparable(<Approx, _Int1, _Int2>, <Exact, _Int1*, _Int2*>) = true
[comp8] comparable(<Exact, _Int1, _Int2>, <Approx, _Int1*, _Int2*>) = true

/* rest niet specificeren -> alle andere combinaties leveren !- true (en niet false) */

[mbstr] must-be-string(_TVListDescr, _ValueExpr) =
  OK when ve-check(_TVListDescr, _ValueExpr) = <OK, <String, _Int1, _Int2>>
[mbstr2] must-be-string(_TVListDescr, _ValueExpr) =
  Type of _ValueExpr must be a string!
  when ve-check(_TVListDescr, _ValueExpr) = <OK, <_TypDescr, _Int1, _Int2>>,
  _TypDescr != String
[mbstr3] must-be-string(_TVListDescr, _ValueExpr) =
  Error when ve-check(_TVListDescr, _ValueExpr) = <Error, _DTDescr>,
  _Error != OK

[mbl1] must-be-length1(_TVListDescr, _ValueExpr) =
  OK when ve-check(_TVListDescr, _ValueExpr) = <OK, <_TypDescr, _Int1, _Int2>>,
  _Int1 = 1
[mbl2] must-be-length1(_TVListDescr, _ValueExpr) =
  Value _ValueExpr must be a single char!
  when ve-check(_TVListDescr, _ValueExpr) = <OK, <_TypDescr, _Int1, _Int2>>,
  _Int1 != 1

end TCPredSem
module TC_Query_sem

imports

TC_QueryPred_syn

exports

context-free syntax

"Table Ref" TableName "exposed ambiguous!" -> ERROR

hiddens

context-free syntax

get-column-descs "(" EnvDescr "," TVListDescr ")" -> CollistDescr

get-column-descs "(" EnvDescr "," TVDescr ")" -> CollistDescr

overlay-columns "(" IdentList "," CollistDescr ")" -> CollistDescr

t-occurs "(" Ident "," TVListDescr ")" -> BOOL

table-id "(" TVDescr ")" -> Ident

equations

[query1] query-check (EnvDescr, SELECT ALL _SelectList _TableExpr) =
  <Error1 AND THEN Error2, 0, CollistDescr>
  when texpr-check (EnvDescr, [] _TableExpr) = <Error1, TVListDescr>,

[query2] query-check (EnvDescr, SELECT DISTINCT _SelectList _TableExpr) =
  <Error1 AND THEN Error2, 0, CollistDescr>
  when texpr-check (EnvDescr, [] _TableExpr) = <Error1, TVListDescr>,

[query3] query-check (EnvDescr, SELECT _SelectList _TableExpr) =
  <Error1 AND THEN Error2, 0, CollistDescr>
  when texpr-check (EnvDescr, [] _TableExpr) = <Error1, TVListDescr>,

[texpr1] texpr-check (EnvDescr, TVListDescr, FromClause WhereClause GroupByClause HavingClause) =
  <Error1 AND THEN Error2 AND THEN Error3 AND THEN Error4, TVListDescr>
  when from-check (EnvDescr, FromClause) = <Error1, TVListDescr>,

[texpr2] texpr-check (EnvDescr, TVListDescr, FromClause WhereClause GroupByClause) =
  <Error1 AND THEN Error2 AND THEN Error3, TVListDescr>
  when from-check (EnvDescr, FromClause) = <Error1, TVListDescr>,

[texpr3] texpr-check (EnvDescr, TVListDescr, FromClause WhereClause GroupByClause) =
  <Error1 AND THEN Error2 AND THEN Error3, TVListDescr>
  when from-check (EnvDescr, FromClause) = <Error1, TVListDescr>,

[texpr4] texpr-check (EnvDescr, TVListDescr, FromClause WhereClause) =
  <Error1 AND THEN Error2, TVListDescr>
  when from-check (EnvDescr, FromClause) = <Error1, TVListDescr>,

[texpr5] texpr-check (EnvDescr, TVListDescr, FromClause GroupByClause HavingClause) =
  <Error1 AND THEN Error2 AND THEN Error3, TVListDescr>
  when from-check (EnvDescr, FromClause) = <Error1, TVListDescr>,

[texpr6] texpr-check (EnvDescr, TVListDescr, FromClause GroupByClause) =
  <Error1 AND THEN Error2, TVListDescr>
  when from-check (EnvDescr, FromClause) = <Error1, TVListDescr>,

[texpr7] texpr-check (EnvDescr, TVListDescr, HavingClause) =
  <Error1 AND THEN Error2, TVListDescr>
  when from-check (EnvDescr, FromClause) = <Error1, TVListDescr>,

[texpr8] texpr-check (EnvDescr, TVListDescr, FromClause) =
  <Error, TVListDescr>
  when from-check (EnvDescr, FromClause) = <Error, TVListDescr>
<Error1 AND Error2, [ColDescr*1 ,ColDescr*2]>
when sell~check(EnvDescr, TVListDescr, _ValueExpr) = <Error1, [ColDescr*1]>,
sell~check(EnvDescr, TVListDescr, _ValueExpr) = <Error2, [ColDescr*2]>
[sql15 | sell~check(EnvDescr, TVListDescr, *) =
<OK, get~column~descrs(EnvDescr, TVListDescr)>]
[goda1 | get~column~descrs(EnvDescr, []) = []]
[goda2 | get~column~descrs(EnvDescr, [TVDescr _TVDescr*1]; =
[ColDescr*1 ,ColDescr*2]]
when get~column~descrs(EnvDescr, TVDescr) = [ColDescr*1],
get~column~descrs(EnvDescr, [TVDescr*1]) = [ColDescr*2]
[goda1 | get~column~descrs(EnvDescr, <Ident1, _Ident2, _ColListDescr, _TConsListDescr>) =
_ColListDescr]
[goda2 | get~column~descrs(EnvDescr, <Ident1, _Ident2, _IdentList, _QuerySpec, _Bool>) =
overlay~columns(_IdentList, _ColListDescr)
when query~check(EnvDescr, _QuerySpec) = <Error, _Ident’, _ColListDescr>
[ovlca1] overlay~columns([], []) = []
[ovlca2] overlay~columns([_Ident _Ident*1], [ColDescr _ColDescr*1]) =
[<Ident, _Ident’, _DTDescr, _DefDescr, _CConsListDescr, _Bool> _ColDescr*1]
when _ColDescr = <Ident’, _DTDescr, _DefDescr, _CConsListDescr, _Bool>,
overlay~columns([_Ident*1], [ColDescr*1]) = [ColDescr*1]
[subq1 | subq~check(EnvDescr, TVListDescr, (SELECT ALL _ResultSpec _TableExpr)) =
<Error, A, _ColListDescr>
when subq~check(EnvDescr, TVListDescr, (SELECT _ResultSpec _TableExpr)) =
<Error, _Ident’, _ColListDescr>
[subq2 | subq~check(EnvDescr, TVListDescr, (SELECT DISTINCT _ResultSpec _TableExpr)) =
<Error, D, _ColListDescr>
when subq~check(EnvDescr, TVListDescr, (SELECT _ResultSpec _TableExpr)) =
<Error, _Ident’, _ColListDescr>
[subq3 | subq~check(EnvDescr, TVListDescr, (SELECT _ValueExpr _TableExpr)) =
<Error1 AND THEN Error2, X, _ColListDescr>
when temp~check(EnvDescr, TVListDescr, _TableExpr) = <Error, _TVListDescr>,
sell~check(EnvDescr, TVListDescr, _ValueExpr) = <Error2, _ColListDescr>
[subq4 | subq~check(EnvDescr, TVListDescr, (SELECT ALL * _TableExpr)) =
subq~check(EnvDescr, TVListDescr, (SELECT * _TableExpr)) =
<Error1 AND THEN Error2, X, _ColListDescr>
when temp~check(EnvDescr, TVListDescr, _TableExpr) = <Error, _TVListDescr>,
sell~check(EnvDescr, TVListDescr, *) = <Error2, _ColListDescr>
end TC_Query_sen
module TC_QueryPred

imports
  TC_Pred_sem
  TC_Query_sem

exports

context-free syntax
  tc (" $EnvDescr ", " $QuerySpec ") -> ERROR

equations
  tc(EnvDescr, QuerySpec) = _Error
    when query-check(EnvDescr, QuerySpec) = <_Error, _Ident, _ColListDescr>

end TC_QueryPred

module TypeCheck

imports
  Error
  SchemaCheck
  TC_QueryPred

exports

context-free syntax
  tc (" $Environment ", " $QuerySpec ") -> ERROR

equations
  tc(Environment, QuerySpec) =
    tc(EnvDescr)
      AND THEN
    tc(EnvDescr', QuerySpec)
    when
      env-descr(Environment) = EnvDescr,
      expand-env(EnvDescr, EnvDescr) = EnvDescr'

end TypeCheck
module TR_Schema

imports
Schemas
RDBS

exports

context-free syntax

tr "(" Schema ")" -> SCHEMA
tr "(" SchemaElement ")" -> SCHEMA
tr "(" Indent ")" -> VAR
attributes "(" (TableElement ",")* ")" -> VAR_TUP
domains "(" (TableElement ",")* ")" -> REL
donmas "(" (DataType ")" -> REL
attributes "(" QuerySpec ")" -> VAR_TUP
attributes "(" QuerySpec ")" -> VAR_TUP

equations

[ 1] tr CREATE SCHEMA AUTHORIZATION _Ident =
   schema

[ 1] tr CREATE SCHEMA AUTHORIZATION _Ident _SchemaElement _SchemaElement* =
   schema _RDef1 _RDef2
   when tr _SchemaElement = schema _RDef1,
   tr CREATE SCHEMA AUTHORIZATION _Ident _SchemaElement* = schema _RDef2

[ 2] tr CREATE TABLE _Ident1 _Ident2 ( _TableElement* ) =
   schema tr (_Ident2) =
   ( attributes _TableElement* ) & domains _TableElement*

[ 2] tr CREATE TABLE _Ident1 _Ident2 ( _TableElement* ) =
   schema tr (_Ident2) =
   ( attributes _TableElement* ) & domains _TableElement*

[ 2] tr CREATE VIEW _Ident1 _Ident2 ( _ViewColumnList ) AS QuerySpec WITH CHECK OPTION =
   schema tr (_Ident2) =
   ( attributes _QuerySpec ) & domains _QuerySpec

[ 2] tr CREATE VIEW _Ident1 _Ident2 ( _ViewColumnList ) AS QuerySpec WITH CHECK OPTION =
   schema tr (_Ident2) =
   ( attributes _QuerySpec ) & domains _QuerySpec

[ 2] tr CREATE VIEW _Ident1 _Ident2 _AS QuerySpec =
   schema tr (_Ident2) =
   ( attributes _QuerySpec ) & domains _QuerySpec

[ 2] tr CREATE VIEW _Ident1 _Ident2 ( _ViewColumnList ) AS QuerySpec WITH CHECK OPTION =
   schema tr (_Ident2) =
   ( attributes _QuerySpec ) & domains _QuerySpec

[ 2] tr CREATE VIEW _Ident1 _Ident2 ( _ViewColumnList ) AS QuerySpec =
   schema tr (_Ident2) =
   ( attributes _QuerySpec ) & domains _QuerySpec

[ 2] tr CREATE VIEW _Ident1 _Ident2 _AS QuerySpec =
   schema tr (_Ident2) =
   ( attributes _QuerySpec ) & domains _QuerySpec

[ 2] tr PRIVILEGEDEF = schema

[ 3] tr Ident (_Char*) = VAR (_Char*)

[ 4] attributes _TableElement _TableElement* =
   < Vars1 _Vars2>
   when attributes _TableElement = < Vars1>
   attributes _TableElement* = < Vars2

[ 4] attributes _Ident _DataType _DefaultClause _ColumnConstraint* =
   < _Ident>

[ 4] attributes _Ident _DataType _ColumnConstraint* =
   < _Ident>

[ 4] attributes _TableConstraintDef =
   < >

[ 5] domains _TableElement _TableElement* =
   domains _TableElement & domains _TableElement*

[ 5] domains _Ident _DataType _DefaultClause _ColumnConstraint* =
   domains _DataType

[ 5] domains _Ident _DataType _ColumnConstraint* =
   domains _DataType

[ 6] domains _StringType = Strings
[ 6] domains _ExactType = Integers
[ 6] domains _ApproxType = Reals

[ 7] attributes _Ident _ColumnNames =
   < _Var _Vars>
   when tr _Ident = _Var
   attributes _ColumnNames = < _Var

[ 7] attributes _Ident =
   < _Ident>

end TR_Schema
module TR_ValueExpr

import

ValueExpressions
Expressions

exports

context-free syntax

tr "(" valueExpr ")" -> EXPR

equations

[vel ] tr(ValueExpr + _Term) = tr(ValueExpr) + tr(_Term)
[vem ] tr(ValueExpr - _Term) = tr(ValueExpr) - tr(_Term)
[vw3 ] tr(_Term * _Factor) = tr(_Term) * tr(_Factor)
[vw4 ] tr(_Term / _Factor) = tr(_Term) / tr(_Factor)
[vw5 ] tr(+ _Primary) = + tr(_Primary)
[vw6 ] tr(- _Primary) = - tr(_Primary)
[vw7 ] tr(ParameterName INDICATOR ParameterName') = null

% case ParameterName: see case ColumnSpec
[vw8 ] tr(_EmbeddedVarName INDICATOR _EmbeddedVarName') = null
[vw9 ] tr(_EmbeddedVarName) = null
[vw10] tr(stringlit(" " _Char+ "")) = str_const(" " _Char+ "")
[vw11] tr(exactlit(_Char+)) = real_const(_Char+)
[vw12] tr(approxlit(_Char+)) = real_const(_Char+)
[vw13] tr(USER) = 'lit'
[vw14] tr(ident(_Char1).ident(_Char2)) = var(_Char2)
[vw15] tr(ident(_Char1)) = var(_Char+)
[vw16] tr(_ValueExpr) = tr(_ValueExpr)
[vw17] tr(_SetFuncSpec) = null

end TR_ValueExpr


module TR_QueryFred

import

TR_ValueExpr
Predicates
QuerySpecs
Expressions

exports

context-free syntax

tr "!' Predicate ")" -> EXPR
tr "!' QuerySpec ")" -> REL

dmods

context-free syntax

tr sel "(" SelectList ")" -> EXPR_TUP
tr refs "(" (TableReference ",")+ ")" -> REL
tr cte "(" TableReference ")" -> VAR
tr ach "(" SearchCondition ")" -> EXPR
tr subq "(" SubQuery ")" -> REL

equations

[pred1] tr(_ValueExpr1 = _ValueExpr2) = (tr(_ValueExpr1) = tr(_ValueExpr2))
[pred2] tr(_ValueExpr1 <> _ValueExpr2) = (tr(_ValueExpr1) <> tr(_ValueExpr2))
[pred3] tr(_ValueExpr1 < _ValueExpr2) = (tr(_ValueExpr1) < tr(_ValueExpr2))
[pred4] tr(_ValueExpr1 > _ValueExpr2) = (tr(_ValueExpr1) > tr(_ValueExpr2))
[pred5] tr(_ValueExpr1 <= _ValueExpr2) = (tr(_ValueExpr1) <= tr(_ValueExpr2))
[pred6] tr(_ValueExpr1 >= _ValueExpr2) = (tr(_ValueExpr1) >= tr(_ValueExpr2))
[pred7] tr(_ValueExpr1 _CompOp _SubQuery) = tr(_ValueExpr1 _CompOp ALL _SubQuery)
[pred8] tr(_ValueExpr NOT BETWEEN _ValueExpr2 AND _ValueExpr3) = 

tr(_ValueExpr1 BETWEEN _ValueExpr2 AND _ValueExpr3)
[pred9] tr(_ValueExpr1 BETWEEN _ValueExpr2 AND _ValueExpr3) = (tr(_ValueExpr1) >= tr(_ValueExpr2)) 

& (tr(_ValueExpr1) <= tr(_ValueExpr3))
[pred10] tr(_ValueExpr1 _SubQ) = tr(_ValueExpr1) 

& tr-subq(_SubQuery)
[pred11] tr(_ValueExpr1 _ValueSpec) = 

tr(_ValueExpr1) 

& tr(_ValueSpec)
[pred12] tr(_ValueExpr1 _ValueSpec1, _ValueSpec2) = 

tr(_ValueExpr1) 

& tr(_ValueSpec1, _ValueSpec2)
[pred13] tr(_ValueExpr1 NOT IN _SubQuery) = 

tr(_ValueExpr1) 

& tr(_SubQuery)
[query1] tr(SELECT ALL _SelectList TableExpr) =
tr(SELECT _SelectList TableExpr)

[query2] tr(SELECT DISTINCT _SelectList TableExpr) =
| tr(SELECT _SelectList TableExpr)

[query3] tr(SELECT _SelectList FROM TableRefList WHERE _SearchCondition) =
| { tr-sell(_SelectList) <= $ tr-ref (_TableRefList) | tr-arch(_SearchCondition) } when _SelectList != *

[query4] tr(SELECT _SelectList FROM TableRefList) =
| { tr-sell(_SelectList) <= $ tr-ref (_TableRefList) | true } when _SelectList != *

[query5] tr(SELECT * FROM TableRefList WHERE _SearchCondition) =
| { $ tr-ref (_TableRefList) | true } when _SelectList != *

[sell1] tr-sell(_ValueExpr, _ValueExpr) =
Expr, Expr when _ValueExpr = Expr,
tr-sell(_ValueExpr) = Expr

[tref1] tr-tref(ident, _TableRefList) =
| tr-tref(_Ident, _TableRefList) | true when _TableRefList != _Ident

[tarch1] tr-arch(_SearchCondition OR BooleanTerm) =
tr-arch(_SearchCondition) | tr-arch(BooleanTerm)

[tarch2] tr-arch(BooleanTerm AND BooleanFactor) =
tr-arch(BooleanTerm) & tr-arch(BooleanFactor)

[tarch3] tr-arch(NOT BooleanPrimary) =
| tr-arch(BooleanPrimary)

[tarch4] tr-arch(_Predicate) =
null

[tarch5] tr-arch(_SearchCondition) =
tr-arch(_SearchCondition)

[subq1] tr-subq((SELECT ALL _ValueExpr _TableExpr)) =
| tr(SELECT _ValueExpr _TableExpr)

[subq2] tr-subq((SELECT DISTINCT _ValueExpr _TableExpr)) =
| tr(SELECT _ValueExpr _TableExpr)

end TR_QueryPred