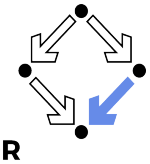
RISC



RESEARCH INSTITUTE FOR
SYMBOLIC COMPUTATION

JKU

JOHANNES KEPLER
UNIVERSITY LINZ

Semantics-Based Rapid Prototyping of a Subset of SQL

Wolfgang Schreiner, William Steingartner

February 2025

RISC Report Series No. 25-02

ISSN: 2791-4267 (online)

Available at `https://doi.org/10.35011/risc.25-02`

Editors: RISC Faculty
B. Buchberger, R. Hemmecke, T. Kutsia, G. Landsmann, P. Paule, V. Pillwein, N. Popov, S. Radu, J. Schicho, C. Schneider, W. Schreiner, W. Windsteiger, F. Winkler.

Supported by: Aktion Österreich–Slowakei project 2024-05-15-001, KEGA project 030TUKE-4/2023

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Semantics-Based Rapid Prototyping of a Subset of SQL*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

William Steingartner
Department of Computers and Informatics
Technical University of Košice, Slovak Republic
William.Steingartner@tuke.sk

February 21, 2025

Abstract

This report documents the application of our semantics-based language generator SLANG to developing a rapid prototype of a non-trivial domain-specific language, a substantial subset of the Structured Query Language SQL that we have named SubSQL. After developing a mathematical/logical formulation of the language's abstract syntax, formal type system, and denotational semantics, we have translated this formulation into a SLANG specification from which the SLANG software generates Java code that implements a parser, a printer, a type-checker, and an executor of the language. This implementation is based on several manually created Java classes that implement the mathematical domains and operations used in the formalization, a simple persistent database, and a high-level application programming interface that allows to execute complete SubSQL scripts from file or individual SubSQL commands within Java programs. The results represent a blueprint for the semantics-based development of other domain-specific languages of similar complexity.

*Supported by the Aktion Österreich–Slowakei project 2024-05-15-001 "Formalizing and Generating Executable Implementations of Domain-Specific Languages" and by the KEGA project 030TUKE-4/2023 "Application of new principles in the education of IT specialists in the field of formal languages and compilers" granted by the Cultural and Education Grant Agency of the Slovak Ministry of Education.

Contents

1 Introduction

In this report, we document the use of the "semantics-based language generator" SLANG [8, 10] to implement a rapid prototype of a substantial subset of the database language SQL [14], based on a formalization of the language's concrete syntax, abstract syntax, type system, and denotational semantics. From this formalization, the SLANG software generates a set of classes in Java 21 that implement a parser, a printer, a type-checker, and an executor of the language. The classes are connected to a simple implementation of an in-memory database that is made persistent via the opencsv library [2]. An API allows to embed individual SQL commands into Java code or execute SQL scripts from the command line. The functionality of the rapid prototype has been experimentally validated (but not thoroughly tested) on a sample database.

We have developed SLANG with the main goal to simplify the rapid prototyping of domain-specific languages (DSLs) [15, 4, 5]. DSLs have naturally arisen in a multitude of diverse application domains in order to allow the programmatic solution of problems in these domains on a much higher level of abstraction than is possible in general-purpose programming languages. In the past, we have demonstrated our ideas by formalizing in SLANG simple languages, such as the educational programming language JANE [11], the educational robot control language Karel [13, 12, 6], and the extended finite state machine language EFSM, which has been used to prototype the machine controller of a robotic vacuum cleaner [9].

The goal of the present paper is to demonstrate that this approach also scales to a larger DSL that is actually used in industrial practice and has a substantially more complex syntax and semantics. This DSL is the Structured Query Language (SQL), which since the 1970s has been the de-facto standard in relational database management systems, and has a variety of different implementations. However, while there exists an official ISO/IEC standard for SQL [14], this standard is overwhelmingly complex. Therefore real SQL implementations often implement only subsets of this standard and, even within these subsets, differ from the standard to varying degrees; thus, in effect, different SQL implementations are incompatible with each other. Furthermore, the standard is not freely available, such that implementers often resort to freely available resources, such as white papers, tutorials, manuals, etc., with varying degrees of exactness, completeness, and correctness. All of this makes it in fact quite difficult to state what the "real SQL" is.

We have based our work on various such free resources, from the W3 Schools tutorial on SQL [7] (which is a good starting point but provides mainly examples, rather than a general description of SQL), over the documentation of the SQLite library [1] (which gives a comprehensive description of the SQL syntax in the form of syntax diagrams, but not a correspondingly detailed description of the SQL semantics), to the reference manual of the popular database management system MySQL [3] (which contains a more detailed explanation of the semantics in natural language which is, however, quite informal and leaves many questions open).

From these sources we have derived a (quite restricted but still substantial) subset of SQL that we have appropriately called "SubSQL". This SQL dialect supports the following features:

- The SQL data definition commands `CREATE TABLE` and `DROP TABLE`.
- The SQL data manipulation commands `INSERT INTO`, `UPDATE`, and `DELETE`.

- The SQL data query command `SELECT` with the clauses `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`.
- Subqueries, i.e., `SELECT` queries nested inside other `SELECT` queries (at arbitrary depth) where inner queries may refer to variables from outer queries.
- The SQL aggregate functions `MIN()`, `MAX()`, `COUNT()`, `SUM()` and `AVG()` that reduce a table column to a single value.
- The combination of tables by `CROSS JOIN` (short: `","`), `(INNER) JOIN`, `LEFT (OUTER) JOIN`, `RIGHT (OUTER) JOIN`, `FULL (OUTER) JOIN`.
- Values of the integer type `INT(`n`)` and the string type `VARCHAR(`n`)` (where the parameter n, however, does actually not limit the size of the value but only the width of its display).
- `NULL` values and a 3-valued logic that includes the truth value `NULL` (interpreted as "unknown").

Starting with the definition of the abstract syntax of SubSQL, we have elaborated a formal type system (as a logical inference system) for the language and given it a formal semantics (as a set of denotation functions that map syntactic domains to semantic ones). This framework was developed without any tool support (apart from a plain text editor) and iteratively reviewed and revised. Ultimately, it was decided to be good enough to be translated into the specification language of SLANG, accompanied by manually crafted Java classes that implement the mathematical domains used by the type system and the denotational semantics, as well as Java classes that implement a basic database and a simple API for controlling this database via individual SubSQL commands or complete SubSQL scripts. The (manually crafted and automatically generated) code of the rapid prototype was tested and debugged on a simple database until the outcome validated its correctness to a level of confidence that satisfied us.

However, we would like to emphasize, that this SLANG-based rapid prototype implementation of SubSQL is definitely not ready for use in production. First, the implemented syntax and semantics definitely deviates in some aspects from the SQL standard (from trivial features such as that string literals are separated by double quotes rather than single quotes up to deeper issues such as that the `USING` option of a table join leaves in the resulting table two duplicates of the common column rather than merging them into one). We deemed these and similar issues not important enough to invest more time and effort, since our goal was to develop a proof of concept rather than a full-fledged implementation. Second, the result most likely contains both errors on the semantic level (due to misunderstandings of semantic concepts or their inadequate formulations) and bugs on the implementation level (which were not caught by testing). However, our experiments seem to indicate that the quality of the result is good enough to substantiate our main point, namely that SLANG can indeed be effectively used to develop rapid prototypes of substantial domain-specific languages from the formalization of their syntax, semantics, and type system.

Actually, the point of the type system deserves some high-lighting. While most implementations of SQL are just based on runtime type checking (errors are mostly caught during the execution of SQL commands), our implementation actually provides a static type checker that catches many/most errors without actually executing the commands. Furthermore, commands that

pass the type checking are not just "interpreted", they are actually "compiled" into a semantic representation that can then be executed with little overhead (also multiple times for different table values). Thus our SQL implementation follows well-established principles for the design and implementation of programming languages.

The SLANG specification of SubSQL, along with the generated code and all the supporting files required to run the software, can be downloaded from the following location:

`https://www.risc.jku.at/research/formal/software/SLANG/public/SubSQL.tgz`

All of this may be freely used for own experiments, modifications, and extensions.

The core of this document is Section 2 that briefly outlines the key ideas of the formalization of SubSQL, the main steps of its implementation, and the actual execution of the resulting prototype. Section 3 summarizes our experience and outlines our further plans.

The main part of this document, however, consists of its appendices: Appendix A presents the manually crafted formalization of SubSQL in mathematical and logical style. Appendix B provides the SLANG version of this formalization, from which the parser, printer, type checker, and executor are generated, along with supporting Java code for the type system, the semantics, and the database implementation. This results in a Java API for SubSQL whose use is demonstrated by an example session.

2 SubSQL: A Subset of SQL

In this section, we give a high level overview on the formalization of SubSQL and its implementation. It is mainly intended as a guide through the detailed specification and code that are given in Appendix A and Appendix B; they should be consulted to understand the actual details of the formalization.

2.1 Semantic Model

The abstract syntax of SubSQL is given in Subsection A.1. The basic principles of the formalization of its semantics are best explained by first describing the domain that forms the core of the semantic model:

$$
\begin{aligned}
\mathit{State} &:= \mathit{store}\colon \mathit{Store} \times \mathit{db}\colon \mathit{Database} \times \mathit{tables}\colon \mathit{Tables} \times \mathit{rows}\colon \mathit{Table} \\
\mathit{Store} &:= \mathit{Id} \rightarrow_{\perp} \mathit{Value} \\
\mathit{Database} &:= \mathit{Id} \rightarrow_{\perp} \mathit{Table} \\
\mathit{Tables} &:= \mathit{Table}^{*} \\
\mathit{Table} &:= \mathit{Row}^{*} \\
\mathit{Row} &:= \mathit{Value}^{*}
\end{aligned}
$$

The domain *State* is the core domain of the semantics. The meaning of every syntactic phrase depends on a given element of this domain, the (current) *state*, which is a tuple with the following components:

- The (partial) mapping *store* of identifiers to values determines the meaning of every meta-variable that can be inserted into SQL commands wherever a value is expected. Thus an SQL statement can be parameterized over values, which may become handy when embedding it into a Java program.
- The (partial) mapping *db* of identifiers to tables represents the content of the database on which every SQL statement operates.
- A list of *tables* represents the essential context for the evaluation of every SQL phrase. This list is organized as a stack: its top represents the "current table" on which the phrase operates. A stack of such tables exists rather than just a single one because, in nested `SELECT` queries, each inner query pushes by its `FROM` clause another such table onto the stack. Thus, the depth of the stack indicates the number of nested queries that are currently being evaluated.
- A list of *rows* represents an auxiliary context for the evaluation of some SQL phrases. Again, this list is organized as a stack: its top represents the "current row" of the "current table" on which the phrase operates. Again, a stack of such rows exists rather than a single one because, during the evaluation of nested `SELECT` queries, each query may have its own notion of the current row.

In SQL statements, most identifiers refer to table columns. In order to actually determine the current value of an column reference, we need as the core information a triple $\langle column, depth, pos \rangle$ with the following interpretation:

- *column* is a binary value that indicates whether the identifier refers to an entire table column (which can be determined from the *tables* stack) or just to a single value in a column (which can be determined from the *rows* stack).
- *depth* denotes the nesting depth at which the identifier has been introduced; it therefore also represents the stack depth at which the identifier can find its value (from the *tables* stack or from the *rows* stack).
- *pos* represents the column position denoted by the identifier within the referenced table/row.

For every column reference, the type checker determines by static analysis this triple (see below) and annotates the identifier with this triple; when the phrase is later evaluated for a given state, only this information is used to derive its denotation: if *column* indicates that the identifier denotes a column, this column is determined as $\mathsf{col}(tables[depth], pos)$ (where $\mathsf{col}(t, c)$ denotes column c of table t); analogously, if the identifier denotes a single value, this value is determined as $rows[depth][pos]$.

In the actual formulation of the semantics function given in Subsection A.3, the relevant equation is as follows:

```
⟦Exp⟧: State →⊥ Value
======================
```

```
⟦ColumnRef^{csym,pos,kind,depth}⟧(s) :=
  match kind with
    Kind.cell ⟶ s.rows[depth][pos]
  | Kind.column ⟶
      Value.seq(λi∈domain(s.tables[depth]). s.tables[depth][i][pos])
```

A column reference is annotated with the information mentioned above. Depending on the kind of the reference, the given state s is looked up for the individual value or for a full column, as explained above.

Actually, the type checker determines for a column reference also a symbol *csym* that represents all information needed to type-check the SQL phrase, in particular the type of the values stored in that particular table column. For all of this, the system depends on the core concept of an *environment* which is an element of the following domain *Env*:

$$\begin{aligned} \mathit{Env} := {} & \mathit{db}\colon \mathit{TableEnv} \times \mathit{venv}\colon \mathit{VarEnv} \times \mathit{tenv}\colon \mathit{TableEnv} \times \\ & \mathit{cstack}\colon \mathit{ColumnSymbols}^* \times \mathit{cenv}\colon \mathit{ColumnEnv} \times \\ & \mathit{ckind}\colon (\mathit{ColumnSymbol} \to \mathit{Kind}) \times \mathit{cdepth}\colon (\mathit{ColumnSymbol} \to \mathbb{N}) \times \\ & \mathit{depth}\colon N \times \mathit{gsyms}\colon \mathit{CoumnSymbolSet} \end{aligned}$$

The interpretation of these components is as follows (we omit the definitions of the auxiliary domains which are given in Subsection A.2):

- *db* is a partial mapping of identifiers to table symbols; it represents the static information on all tables in the database.
- *venv* is a partial mapping of identifiers to variable symbols; it represents the static information on the "meta-variables".
- *tenv* is a partial mapping of identifiers to table symbols; it represents the static information on all tables that are visible (can be referenced) in the current context.
- *cstack* is a stack of sequences of column symbols; each sequence represents the static information on the "current table/row". Analogous to the stacks *tables* and *rows* in domain *State*, there exists a stack of such sequences rather than a single sequence because of the possibility of nested `SELECT` queries.
- *cenv* is a partial mapping of identifiers to column symbols; it represents the static information on all columns that are visible (can be referenced) in the current context.
- *ckind* and *cdepth* map every column symbol to its kind (full column or single value) and the depth of *cstack* where it has been introduced.
- *depth* represents the current stack depth (i.e., the number of context nestings).
- *gsyms* represents the set of those symbols that are listed in the `GROUP BY` clause of the current `SELECT` query (if any).

Using this information, the type-correctness of a sequence of SQL statements can be checked and every column identifier can be annotated by the triple $\langle column, depth, pos \rangle$ required by the semantics to determine its meaning for a given state.

In the actual formulation of the type system given in Subsection A.2 the relevant inference rule is as follows:

```
Env ⊢ Exp: exp(ExpType)
=======================
env ⊢ ColumnRef: columnRef(csym)
type = type(csym.type)
kind = env.ckind(csym)
depth = env.cdepth(csym)
pos ∈ ℕ  env.cstack[depth][pos] = csym
grouped = depth < length(env.cstack)-1 ∨ csym ∈ env.gsyms
etype = ⟨type:type, kind:kind, group:grouped⟩
-----------------------------------------------------------
env ⊢ ColumnRef^{csym,pos,kind,depth}: exp(etype)
```

Given the current environment *env*, the inference rule determines by the inference rule for an auxiliary judgment columnRef the symbol *csym* of the denoted column and looks up its *kind* and *depth* in the relevant components of the environment. It then also determines the position *pos* of the symbol in the column sequence at depth *depth* of the stack *cstack* and annotates the column reference with all the derived information. The judgment also determines the type *etype* of the expression represented by the column reference and returns it as a result of the judgment.

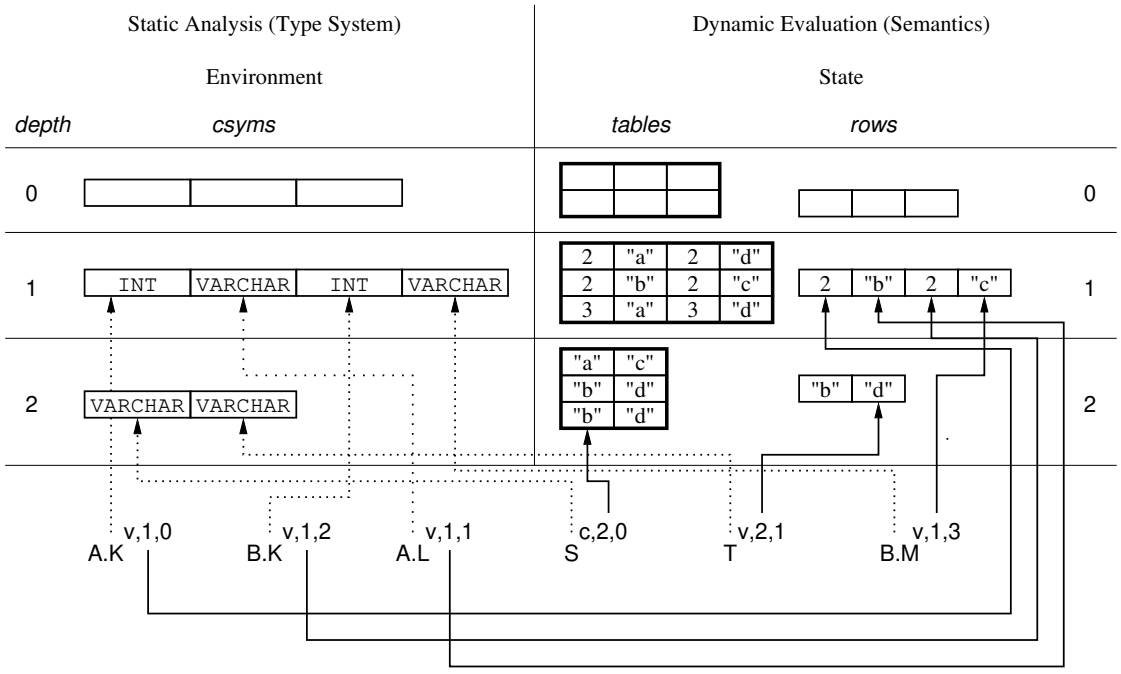
We illustrate the relationship between the environment of the type system (static analysis) and the state of the semantic model (dynamic evaluation) by the following small example:

```
CREATE TABLE A (K INT(3), L VARCHAR(5));
CREATE TABLE B (K INT(3), M VARCHAR(8));
CREATE TABLE C (S VARCHAR(5), T VARCHAR(8));
... (SELECT * FROM A,B WHERE A.K = B.K AND
      A.L = (SELECT MIN(S) FROM C WHERE T = B.M)) ... ;
```

Figure 1 illustrates the static analysis and dynamic evaluation of the stated SELECT query:

- The static analysis associates to each column reference a symbol (these associations are depicted on the left side of the figure by dotted arrows) which indicates the type of the values in this column. This symbol is taken from the stack *csyms*; each element of this stack represents the sequence of symbols of the current table at the corresponding nesting depth.

  If we assume that the query `SELECT * FROM A,B` occurs at depth 1, the table at this depth represents the cross join of tables A and B and the sequence consists of the symbols for columns A.K, A.L, B.K, and B.M. Consequently, the inner query `SELECT MIN(S) FROM C` occurs at depth 2; the symbol sequence at that depth represents the columns C.S (short: S) and C.T (short: T).




```
... (SELECT * FROM A,B WHERE A.K = B.K AND A.L = (SELECT MIN(S) FROM C WHERE T = B.M)) ...
```

Figure 1: Static Analysis and Dynamic Evaluation

- Furthermore, the static analysis annotates column reference with a triple $\langle column, depth, pos \rangle$, which indicates where the dynamic evaluation can look up the value of that reference in the current state (these lookups are depicted on the right side of the figure by solid arrows).

  For instance, the column reference A.K in the query is annotated with the triple ⟨v,1,0⟩ which states that this reference denotes a single value in the row at depth 1 at position 0; thus this value is looked up as $rows[1][0]$. On the other hand, the column reference S in the query `SELECT MIN(S)` (where `MIN` represents an SQL aggregation function that reduces a column to a single value) is annotated with the triple ⟨c,2,0⟩, which indicates that this reference denotes a column in the table at depth 2 at position 0; thus this column is looked up as $\mathsf{col}(tables[2], 0)$.

After this general overview, we will now provide some examples of the rules for the static analysis and semantic evaluation of SubSQL statements.

2.2 Statements

We start with the formalization of an entire SubSQL session and then proceed to formalize the various kinds of statements that make up such a session.

Sessions

The abstract syntax of SubSQL is given in Subsection A.1, its top-level domain is called `Session`. The judgment and inference rule for type-checking this statement are presented in Subsection A.2 as follows:

```
Env ⊢ Session: session(Env)
==========================
env0 ⊢ Stats: stats(env)
------------------------
env0 ⊢ Stats: session(env)
```

A judgment $env_0 \vdash Stats : \mathsf{session}(env)$ checks whether the session *Stats* is well-formed in a given environment env_0; if this is the case, it creates a new environment *env* that encompasses all the information that the type-checker can derive about this session through static analysis.

According to the grammar, a session can be only a statement sequence, thus the judgment has a single inference rule that handles this case by reducing the problem of type-checking a session to the problem of type-checking a statement sequence by the corresponding judgment $env_0 \vdash Stats : \mathsf{stats}(env)$.

The denotational semantics of a session is correspondingly formalized by the following definition of a semantic function:

```
⟦Session⟧: State →⊥ State × Tables
==================================
⟦Stats⟧(s) := ⟦Stats⟧(s)
```

The semantic function for sessions takes the current state *s* (with the initial version of the database) and returns a pair of a new state (which reflects all the updates that have been performed on the database) and a list of tables (which represent the outputs of the `SELECT` queries performed as statements). It does so by delegating this task for a session (that encapsulates a statement sequence) to the corresponding semantic function for statement sequences.

Appendix A describes in detail the type judgments and denotation functions for statement sequences and for generic statements. Here, however, we immediately proceed the formalization of the various concrete types of statements.

`CREATE TABLE`

The judgment and inference rule for type-checking the `CREATE TABLE` statement are as follows:

```
Env ⊢ CreateTableStat: createTableStat(Env)
===========================================
id = toId(ID)  id ∉ domain(env0.tenv)
id,[] ⊢ ColumnDecls: columnDecls(csyms)
tsym = ⟨id:id,csyms:csyms⟩
env = env0 with .tenv := env.tenv[id↦tsym]
-------------------------------------------------------------------
env0 ⊢ "CREATE" "TABLE" ID "(" ColumnDecls ")" ";": createTableStat(env)
```

This rule type-checks the statement in a given environment env_0 and, if the check succeeds, creates a new environment *env*. To do this, it converts the string `ID` derived by the lexical analyzer to an identifier `id` used in the formalization. It then checks that in env_0 this identifier is not yet associated with a table. Next, the identifier is used to create from an empty sequence of column symbols, by checking the declarations of the table columns, the corresponding column symbols *csyms*. From this, a table symbol *tsym* is created, and the new environment *env* is derived from the given environment env_0 by mapping *id* to *tsym*.

The corresponding denotational semantics of this statement is formalized by the following definition of a semantic function:

```
⟦CreateTableStat⟧: State →⊥ State
=================================
⟦"CREATE" "TABLE" ID "(" ColumnDecls ")" ";"⟧(s) :=
  s WITH .db = s.db[ toId(ID) ↦ [] ]
```

This semantic function takes the current state *s* and extends its database component *db* by a mapping of the table identifier to an empty table (i.e., an empty sequence of rows).

DROP TABLE

The judgment and inference rule for type-checking the `DROP TABLE` statement are as follows:

```
Env ⊢ DropTableStat: dropTableStat(Env)
=======================================
id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv
env = env0 with .tenv := env0.tenv\{⟨id,tsym⟩}
-----------------------------------------------
env0 ⊢ "DROP" "TABLE" ID ";": dropTableStat(env)
```

This rule is the dual of the rule for the `CREATE TABLE` statement. It checks whether the given environment contains a symbol for the denoted table; if not, it results in an error. Otherwise, the symbol is removed from the environment.

The semantics function is of similar simplicity:

```
⟦DropTableStat⟧: State →⊥ State
===============================
⟦"DROP" "TABLE" ID ";"⟧(s) := s WITH .db =
  let id = toId(ID) in s with .db = s.db\{ ⟨id,t⟩ | t ∈ Table }
```

The function updates the database component of the current state by removing any table that is associated to the identifier given in the statement.

INSERT INTO

The judgment and inference rule for type-checking the `INSERT INTO` statement are as follows:

```
Env ⊢ InsertStat: insertStat
============================
id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv  env1 = enterTable(env,tsym.csyms)
env1 ⊢ InsertColumnClause: insertColumnClause(csyms1)
env1 ⊢ TableExp: tableExp(csyms2)
columnsMatch(csyms1,csyms2)
-------------------------------------------------------------------------
env0 ⊢
  "INSERT" "INTO" ID^tsym InsertColumnClause^csyms1 TableExp ";": insertStat
```

The inference rule checks whether in a given environment env_0 the statement is well-formed. For this, it translates the string *ID* into the identifier *id* and checks whether the environment maps *id* to some table symbol *tsym* (if not, this is an error). If yes, the auxiliary function enterTable pushes the column symbols *tsym.csyms* of that table onto the *csyms* stack of env_0 and increases its *depth* by 1, which yields a new environment env_1. In this environment, the rule type-checks the references to the table columns (for which values are to be inserted) which yields the corresponding (potentially unnamed) column symbols $csyms_1$ (if no such references are given, $csyms_1$ equals *tsym.csyms*). Then the rule type-checks the well-formedness of the expression that denotes the values to be inserted, which (if successful) results in column symbols $csyms_2$. Then the rule applies the auxiliary predicate columnsMatch to check whether the symbols $csyms_1$ and $csyms_2$ are compatible with respect to types and (potential) names. Upon success, the statement is annotated with the symbols *tsym* and $csyms_1$.

The definitions of the auxiliary operations enterTable and columnMatch are given in Subsection A.2.

The corresponding semantics function is as follows:

```
⟦InsertStat⟧: State →⊥ State
============================
⟦"INSERT" "INTO" ID^tsym InsertColumnClause^csyms TableExp ";"⟧(s) :=
  let table = s.db[tsym.id] in
  let s0 = newTable(s,table) in
  let rows1 = ⟦TableExp⟧(s0) in
  let rows2 = tableRows^tsym_csyms(rows1) in
  let rows3 = table∘rows2 in
  if ¬satisfiedConstraints(tsym,rows3))
    then ⊥
    else s with .db = s.db[tsym.id↦rows3]
```

This function takes the current state *s* and (potentially) returns a new state that is identical to the current one except that the denoted table has been updated by the insertion of additional values. For this, it takes the table symbol *tsym* with which the type-checker has annotated the

statement and looks up in the database component of s the table *table* with the identifier of the symbol. It then applies the function newTable to create a new state s_0 that pushes *table* onto the stack *tables* of s and then evaluates the table expression in s_0, which leads a sequence of rows $rows_1$. However, the order of the values in these rows does not necessarily correspond to the order of the columns in *table* and it may not contain values for all of its columns. Therefore the semantic function applies the auxiliary function tableRows which uses *tsym* and also the column symbols *csyms* (with which the type checker has annotated the statement) in order to suitably permute the values and add `NULL` values for those columns for which no values are given. This leads the permuted/extended rows $rows_2$ which are added to *table* to give the new rows $rows_3$. Then the semantic function applies the auxiliary predicate satisfiedConstraints to check whether the insertion of the new rows would violate any constraints declared at the table (such as `NON NULL` or `UNIQUE`). If a violation is detected, the result is the undefined state (which indicates a runtime error). If the insertion is admissible, the database component of s is accordingly updated.

The definitions of the auxiliary operations newTable, tableRows, and satisfiedConstraints are given in Subsection A.3.

UPDATE

The judgment and inference rule for type-checking the `UPDATE` statement are as follows:

```
Env ⊢ UpdateStat: updateStat
============================
id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv  env1 = enterTable(env,tsym.csyms)
env1 ⊢ WhereClause: whereClause
env2 = enterRow(env1)
env2,tsym.csyms,∅ ⊢ Assignments: assignments(ids)
-------------------------------------------------------------------
env ⊢ "UPDATE" ID^tsym "SET" Assignments WhereClause ";": updateStat
```

The first part of the rule is similar to that of the `INSERT INTO` statement by creating a new environment that has the column symbols of the denoted table pushed onto the *cstack* component of the environment. In that environment, the well-formedness of the optional `WHERE` clause is checked. Then the auxiliary function enterNew is applied that updates the *ckind* component of the environment in order to map every column symbol of the current table to kind "single value" (which indicates in the type system a switch from the "full table" context to the "individual row" context). The assignments are type-checked within this new environment, which extends the empty set of identifiers to the set *ids* of identifiers that are updated by the assignments (this set is only used within the corresponding judgment to make sure that no identifier is assigned twice).

The definitions of the auxiliary operations enterTable and enterRow are given in Subsection A.2.

The corresponding semantics function is as follows:

```
⟦UpdateStat⟧: State →⊥ State
============================
⟦"UPDATE" ID^tsym "SET" Assignments WhereClause ";"⟧(s) :=
  let table = s.db[tsym.id] in
```

```
let s0 = newTable(s,table) in
let table0 =
  λi∈domain(table).
    let row = table[i] in
    if ¬⟦WhereClause⟧(s0)(row) then
      row
    else
      let s1 = newRow(s0,row) in
      ⟦Assignments⟧(s1,row)
 in if ¬satisfiedConstraints(tsym,table0))
  then ⊥
  else s with .db[tsym.id↦table0]
```

This function uses the table symbol *tsym* (with which the statement has been annotated) to look up the corresponding table in the database component of the given state s. Then s is updated to a new state s_0, by calling the auxiliary operation newTable that pushes this table on the stack *tables* of s. Then the new table $table_0$ is created by processing every *row* of *table* as follows: the predicate denoted by the optional `WHERE` clause is evaluated for state s_0 and *row*. If the predicate does not hold, the row is left unchanged. However, if it does hold, a new state s_1 is constructed by applying the function newRow that pushes *row* onto the *rows* stack of the state (which indicates in the semantics a switch from the "full table" context to the "individual row" context, similarly how the function enterRow worked in the type system). The assignments are performed in this new state on the given row, yielding the updated row. As for the `INSERT` statement, it is checked whether the resulting table $table_0$ satisfies the declared constraints. If it does, the database component of the state is updated with $table_0$.

The definitions of the auxiliary operations newTable and newRow are given in Subsection A.3.

DELETE

The judgment and inference rule for type-checking the `DELETE` statement are as follows:

```
Env ⊢ DeleteStat: deleteStat
============================
id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv  env1 = enterTable(env,tsym.csyms)
env1 ⊢ WhereClause: whereClause
-------------------------------------------------
env: "DELETE" "FROM" ID^tsym WhereClause ";": deleteStat
```

The prerequisites of the rule are exactly the same as those in the first part of the `UPDATE` statement and are explained there.

The semantic function of the statement is as follows:

```
⟦DeleteStat⟧: State →⊥ State
============================
⟦"DELETE" "FROM" ID^tsym WhereClause ";"⟧(s) :=
```

```
let table = s.db[tsym.id] in
let s0 = newTable(s,table) in
let is = { i ∈ domain(table) |
  let row = table[i] in
  ¬⟦WhereClause⟧(s0)(row) } in
let table0 = select(table,is) in
s with .db[tsym.id↦table0]
```

Also the definition of this semantic function is similar to that of the UPDATE statement; the main difference lies in the derivation of the new table content $table_0$. For this, the set *is* of indices is determined that identifies those rows that do *not* satisfy the condition of the WHERE clause. The auxiliary function select is then applied to extract from the table the sequence of rows with those indices; this yields the new content $table_0$ which is used to update the database component of the given state.

SELECT

SubSQL supports a "select statement" (an element of domain *SelectStat*) which is represented by a "table expression" (an element of domain *TableExp*). This expression in turn can be a "select expression" (an element of domain *SelectExp*) for which the judgment and inference rule for type-checking are as follows:

```
Env ⊢ SelectExp: selectExp(ColumnSymbols)
=========================================
env0 ⊢ FromClause: fromClause(env1)
env1 ⊢ WhereClause: whereClause
env1 ⊢ GroupByClause: groupByClause(env2)
env2 ⊢ HavingClause: havingClause
env2 ⊢ SelectClause: selectClause(env3,csyms)
env3 ⊢ orderByClause
-------------------------------------------------------------
env0 ⊢
  "SELECT" DistinctClause SelectClause FromClause WhereClause
   GroupByClause HavingClause OrderByClause ";": selectExp(csyms)
```

This rule demonstrates the flow of information when type-checking the various clauses of a SELECT statement. Starting with the given environment env_0, the optional FROM clause is checked, producing an environment env_1, which in essence captures the "current table" in its *cstack* component. In this environment, the optional WHERE clause and the optional GROUP BY clause are checked. From the latter, we derive an environment env_2 which captures in its *gsyms* component the grouped column symbols. Next, in this environment, the optional HAVING clause and the elements of the SELECT clause are checked. The latter determines a new environment env_3 which essentially reduces the current table to the selected columns; their symbols *csyms* are determined as the result of the judgment. Finally, in this last environment, the optional ORDER BY clause is checked.

The corresponding semantic function mimics this information flow:

```
⟦SelectExp⟧: State →⊥ Table
==========================
⟦"SELECT" DistinctClause SelectClause FromClause^csyms0 WhereClause
          GroupByClause HavingClause OrderByClause ";"⟧(s) :=
  let table1 = ⟦FromClause⟧(s) in
  let s0 = newTable(s,table1) in
  let table2 = ⟦WhereClause⟧(s0,table1) in
  let tables3 = ⟦GroupClause⟧(table2) in
  let tables4 = ⟦HavingClause⟧(s0,tables3)) in
  let table5 = ⟦SelectClause⟧(s0,tables4) in
  let table6 = ⟦DistinctClause⟧(allRows,table5) in
  let table7 = ⟦OrderByClause⟧(table6) in
  table7
```

Given the current state s, the function evaluates the semantic function of the `FROM` clause to determine the initial table $table_1$ that is pushed by application of the auxiliary function newTable onto the *tables* component of s, which yields a new state s_0. Then the semantic function of the `WHERE` clause is evaluated in that state, reducing $table_1$ to a new table $table_2$ which retains only the rows that satisfy the condition specified by the clause. Next, the semantic function for the `GROUP BY` clause is evaluated, which decomposes $table_1$ into a sequence of tables $tables_3$. The application of the semantic function for the `HAVING` clause filters this list, retaining only those tables that satisfy the condition denoted by the clause, yielding a new list $tables_4$. Subsequently, the application of the semantic function for the `SELECT` clause combines all these tables to a single table $table_5$ that, however, only retains those columns that are denoted by the clause. The application of the semantics function of the optional `DISTINCT` clause potentially removes duplicated rows (the parameter *allRows* indicates that the default is to preserve duplicates), which yields a new table $table_6$. Finally, the application of semantic function for the optional `ORDER BY` clause sorts the rows in this table; this yields the final table $table_7$ that is also the result of the semantic function for the `SELECT` statement.

As for the typing rules and semantic functions of the various clauses of the statement, we refer to Subsection A.2 and Subsection A.3.

2.3 Implementation

The formalization of SubSQL which has been outlined in the previous sections and is given in detail in Appendix A represents the basis for the SLANG specification which is given in file `SubSQL.txt` listed in Subsection B.1. The syntax of SLANG is quite closely aligned to the syntax used in the previous section, see [8] for details. For instance, the inference rule for the type-correctness of column references and its semantic function (both given in Subsection 2.1) are as follows:

```
judgement #Env# ⊢ Exp: exp(#ExpType#)
{
```

```
  ...
  inference env ⊢ ExpRef[ColumnRef,Annotation]: exp(etype)
  {
    env ⊢ ColumnRef: columnRef(csym);
    kind: #Kind# = #env.ckind().get(csym)#;
    depth: #Integer# = #env.cdepth().get(csym)#;
    pos: #Integer# = #env.cstack().get(depth).list().indexOf(csym)#;
    type: #Type# = #csym.ctype().type()#;
    # boolean grouped = depth < env.cstack().size()-1 ||
        env.gsyms().contains(csym); #
    etype = #new ExpType(type, kind, grouped)#;
    # Annotation.setColumnInfo(csym,pos,kind,depth); #
  }
  ...
}

function ⟦Exp⟧: #State# → #Value#
{
  equation ⟦ExpRef[ColumnRef,Annotation]⟧(s) = v
  {
    #
      Integer pos = Annotation.getColumnPos();
      Kind kind = Annotation.getColumnKind();
      Integer depth = Annotation.getColumnDepth();
    #
    v = #switch (kind) {
           case Kind.cell -> s.rows().list().get(depth).list().get(pos);
           case Kind.column -> {
            List<Value> values = new ArrayList<Value>();
            for (Row row : s.currentTable().list())
              values.add(row.list().get(pos));
            yield new Value.Seq(new Values(values));
            }
          }#;
  }
  ...
}
```

Likewise, the typing rule and the semantic equation for the core of the SELECT statement (both given in Subsection 2.2) are as follows:

```
judgement #Env# ⊢ SelectExp: selectExp(#ColumnSymbols#)
{
  inference env0 ⊢ TheSelectExp[DistinctClause,SelectClause,FromClause,
```

```
    WhereClause,GroupByClause,HavingClause,OrderByClause]: selectExp(csyms)
  {
    env0 ⊢ FromClause: fromClause(env1);
    env1 ⊢ WhereClause: whereClause;
    env1 ⊢ GroupByClause: groupByClause(env2);
    env2 ⊢ HavingClause: havingClause;
    env2 ⊢ SelectClause: selectClause(env3,csyms);
    env3 ⊢ OrderByClause: orderByClause;
  }
}

function ⟦SelectExp⟧: #State# → #Table#
{
  equation ⟦TheSelectExp[DistinctClause,SelectClause,FromClause,
    WhereClause,GroupByClause,HavingClause,OrderByClause]⟧(s) = table
  {
    table1 = ⟦FromClause⟧(s);
    s0: #State# = #s.newTable(table1)#;
    table2 = ⟦WhereClause⟧(s0,table1);
    tables3 = ⟦GroupByClause⟧(table2);
    tables4 = ⟦HavingClause⟧(s0,tables3);
    table5 = ⟦SelectClause⟧(s0,tables4);
    allRows: #Function<Table,Table># = #(Table t)->t.allRows()#;
    table6 = ⟦DistinctClause⟧(allRows,table5);
    table = ⟦OrderByClause⟧(table6);
  }
}
```

The mathematical domains and operations used by the type system and the semantics are implemented by the Java classes given in Subsection B.2 and Subsection B.3 using the types of the Java collections framework. For instance, the type *Env* of the type system and its associated operations are implemented by the class `subsql.TypeSystem.Env` sketched below:

```
package subsql;
...
public class TypeSystem
{
  ...
  public static record Env(TableEnv db, VarEnv venv, TableEnv tenv,
    List<ColumnSymbols> cstack, ColumnEnv cenv,
    Map<ColumnSymbol,Kind> ckind, Map<ColumnSymbol,Integer> cdepth,
    int depth, Set<ColumnSymbol> gsyms
  )
  {
```

```
    // operations
    ...
    public Env enterTable(ColumnSymbols csyms)
    {
      ColumnEnv cenv0 = cenv.addColumns(csyms);
      return enterTable(cenv0, csyms);
    }
    ...
  }
}
```

Similarly, the type *State* of the semantics and its associated operations are implemented by the class `subsql.Semantics.State` sketched below:

```
package subsql;
...
public class Semantics
{
  ...
  public static record State(PrintStream out,
    Store store, Database db, Tables tables, Table rows)
  {
    // operations
    ...
    public State newTable(Table table)
    {
      List<Table> list = new ArrayList<Table>(tables.list);
      list.add(table);
      return new State(out, store, db, new Tables(list), rows);
    }
    ...
  }
}
```

In the semantic model, the database is represented by the type *Database*; the corresponding Java type is the interface `database.Database`, see Subsection B.4:

```
package database;
...
public interface Database
{
  public void read(String path) throws IOException;
  public void write(String path) throws IOException;
  public void createTable(String name, Info info);
  public void dropTable(String name);
```

```
  ...
}
```

This interface is implemented by a class `database.DatabaseClass` that uses the opencsv library [2] to make the database persistent in the form of text files with embedded CSV content.

From the SLANG specification file `SubSQL.txt`, the SLANG tool generates the Java code for the parser, printer, type-checker, and semantic executor:

```
SLANG Semantics-Based Language Generator 1.0.3 (January 10, 2025)
(c) 2023 https://www.risc.jku.at/research/formal/software/SLANG
This is free software distributed under the terms of the GNU GPL.
Execute "SLANG -h" to see the available command line options.
------------------------------------------------------------------
Reading file SubSQL.txt.
..
Generating files in directory /.../lang/subsql.
Generating code class file SubSQL.java.
Generating ANTLR4 grammar SubSQL.g4.
Generating parser class file SubSQL_parser.java.
Generating for domain Session interface file Session.java.
Generating for domain Stats interface file Stats.java.
Generating for domain Stat interface file Stat.java.
Generating for domain CreateTableStat interface file CreateTableStat.java.
..
Generating for operation session[Session] class file Session_session.java.
Generating for operation stats[Stats] class file Stats_stats.java.
Generating for operation stat[Stat] class file Stat_stat.java.
Generating for operation createTableStat[CreateTableStat] class file
  CreateTableStat_createTableStat.java.
...
Generating for operation session[Session] class file Session_session.java.
Generating for operation stats[Stats] class file Stats_stats.java.
Generating for operation stat[Stat] class file Stat_stat.java.
Generating for operation createTableStat[CreateTableStat] class file ...
...
Generating for operation ␣[Session] class file Session_.java.
Generating for operation ␣[Stats] class file Stats_.java.
Generating for operation ␣[Stat] class file Stat_.java.
Generating for operation ␣[CreateTableStat] class file ...
...
SUCCESS: execution successfully completed.
```

All in all, from a SLANG file with 3,500 lines of specification (120 KB), 18,600 lines of Java code (590 KB) are generated, i.e., the "semantic expansion factor" is about 5–6.

On top of the generated code, we provide a small API in the class `subsql.SubSQL` that allows processing complete SubSQL scripts from file or individual SubSQL commands embedded in Java programs (see Subsection B.5):

```
package subsql;

public class SubSQL
{
  ...
  // SubSQL <sqlpath> [ <dbpath> ]
  public static void main(String[] args) { ... }

  public SubSQL() { ... }
  public SubSQL(PrintStream out, Semantics.Store store, database.Database db)
  { ...}

  public void init() { ... }
  public void read(String path) throws IOException { ... }
  public void write(String path) throws IOException { ... }

  public void set(String var, String value) { ... }
  public void set(String var, Semantics.Value value) { ... }

  public List<Semantics.Table> execute(File file)
    throws FileNotFoundException { ... }
  public lang.subsql.Session_._Operation session(File file)
    throws FileNotFoundException { ... }
  public List<Semantics.Table> execute(lang.subsql.Session_._Operation session)
  { ... }

  public Semantics.Table execute(String text) { ... }
  public lang.subsql.Stat_._Operation stat(String text) { ... }
  public Semantics.Table execute(lang.subsql.Stat_._Operation stat) { ... }
}
```

On the one hand, this class provides a `main` method that can be invoked with the command line argument *sqlpath*, which executes the SubSQL script in file *sqlpath*. If the optional argument *dbpath* is not given, the SubSQL statements are executed on an empty database and the resulting database is discarded at the end. However, if the argument *dbpath* is given, then the database is initially read from file *dbpath* and ultimately written back to that file.

Subsection B.6 demonstrates the use of this program on a sample script of SubSQL commands and shows the output of the `SELECT` queries.

The `main` function is actually based on a lower-level API that enables more flexible use of SubSQL statements within a Java program. The best way to explain this API is by presenting the actual implementation of `main`:

```
// SubSQL <sqlpath> [ <dbpath> ]
public static void main(String[] args)
{
  try
  {
    if (args.length < 1 || args.length > 2) return;
    String sqlpath = args[0];
    String dbpath = args.length == 2 ? args[1] : null;

    // the execution context
    SubSQL sql = new SubSQL();

    // reading the database from file
    if (dbpath != null) sql.read(dbpath);

    // demonstrate how to use meta-variables
    sql.set("NAME", "Wolfgang");

    // parsing, type-checking, executing the SubSQL commands in file
    sql.execute(new File(sqlpath));

    // demonstrate how to execute individual commands
    // Semantics.Table table = sql.execute(
    //   "SELECT * FROM Persons WHERE FirstName = {NAME};");
    // if (table != null) table.println(new PrintStream(System.out), null);

    // writing the database back to file
    if (dbpath != null) sql.write(dbpath);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

First the program creates an object `sql` of type `SubSQL` which represents the complete execution context (in particular, both the type checking environment and the semantic state). By calling `sql.read(dbpath)`, this context is initialized with a database from file `dbpath`. By calling `sql.write(dbpath)` the database is written to this file.

A call `sql.set(NAME,str)` equips this context with a meta-variable `NAME` whose value is string `str` (also values of other types are possible). This meta-variable can be referenced in the script as `{NAME}` wherever an expression may appear; for each such reference the value `str` is inserted into the script.

By calling `sql.execute(file)`, the SubSQL statements in `file` are parsed, type-checked,

and executed; as a side-effect the results of `SELECT statements` are printed (by default to the standard output stream, but the `sql` object may be also initialized with another stream; in case of the `null` stream, no output is produced).

If we call `sql.execute(str)`, the (single) SubSQL statement in string `str` is parsed, type-checked, and executed. If the statement is a `SELECT`, the table resulting from the execution is returned (otherwise, the result is `null`). In fact, the effect of this `sql.execute(str)` is a combination of two methods which can be also individually executed:

```
var op = sql.stat(str);
Semantics.Table table = sql.execute(op);
```

Here the execution of `sql.stat(str)` parses and type-checks the statement in string `str` and translates it into an "operation" `op` that denotes the semantics of the statement. By executing `sql.execute(op)` this semantics is evaluated and returns the resulting table (if any). In this way, a SubSQL statement stored in a string may be just processed once, but executed multiple times (without repeating the processing).

3 Conclusions

In this report, we have documented our experience with using our semantics-based language generator SLANG to implementing a non-trivial DSL, a substantial subset of SQL that we have called SubSQL. The core of the implementation is the SLANG specification that formalizes in 3,500 lines (120 KB) the abstract syntax and the concrete syntax of the language and equips it with a formal type system and a denotational semantics. From this, the SLANG software generates 18,600 lines of Java code (590 KB), which represents a "semantic expansion factor" of about 5–6.

The SLANG formalization follows closely a previously drafted specification in traditional mathematical style using manually crafted Java classes (about 1,300 lines of code) for the implementation of the mathematical domains and operations used in the type system and semantics. The implementation is completed by a simple persistent database (about 500 lines of code) and a high-level API (about 150 lines of code) that allows to execute full SubSQL scripts from file or to embed the execution of individual SubSQL command into a Java program.

Of course, the result is just an experimental rapid prototype, not an industrial-strength implementation, of a relational database system; nevertheless it has shown its functionality in various tests. In particular, our experiment has demonstrated that from a formalization of the syntax, type system, and semantics of a non-trivial DSL (developed in a couple of weeks), an actual implementation can be derived in a relatively straight-forward (in another couple of weeks). In the future, we plan to apply the experience to prototypical implementations of further DSLs in other application domains.

References

[1] SQLite Consortium. SQLite Documentation, February 2025. https://www.sqlite.org/docs.html.

[2] Scott Conway and Andrew Rucker Jones. Opencsv Users Guide, January 2025. https://opencsv.sourceforge.net/.

[3] Oracle Corporation. MySQL 9.2 Reference Manual, February 2025. https://dev.mysql.com/doc/refman/9.2/en/.

[4] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Professional Computing Series. Addison-Wesley, 2010. https://martinfowler.com/books/dsl.html.

[5] Terence Parr. *Language Implementation Patterns — Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009. https://pragprog.com/titles/tpdsl/language-implementation-patterns.

[6] Richard E. Pattis, Jim Roberts, and Mark Stehlik. *Karel The Robot: A Gentle Introduction to the Art of Programming*. Wiley, 2nd edition, July 1994. https://www.wiley.com/en-us/Karel+The+Robot%3A+A+Gentle+Introduction+to+the+Art+of+Programming%2C+2nd+Edition-p-9780471597254.

[7] W3 Schools. SQL Tutorial, February 2025. https://www.w3schools.com/sql/.

[8] Wolfgang Schreiner and William Steingartner. The SLANG Semantics-Based Language-Generator — Tutorial and Reference Manual (Version 1.0.*). Technical report 23-13, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, September 2023. doi:10.35011/risc.23-13.

[9] Wolfgang Schreiner and William Steingartner. Semantics-Based Rapid Prototyping of a Machine Controller Language. In Valerie Novitzká and Anikó Szakál, editors, *Informatics 2024, 2024 IEEE 17th International Scientific Conference on Informatics, Poprad, Slovakia, November 13–15, 2024*, pages 348—353. IEEE, 2024. https://informatics.kpi.fei.tuke.sk.

[10] The SLANG Semantics-Based Language Generator, November 2023. https://www.risc.jku.at/research/formal/software/SLANG.

[11] William Steingartner. On some innovations in teaching the formal semantics using software tools. *Open Computer Science*, 11(1):2–11, 2020. doi:10.1515/comp-2020-0130.

[12] William Steingartner and Valerie Novitzká. Operational Semantics in a Domain-Specific Robot Control Language: a Pedagogical Use Case. *Computer Science and Information System*, 21(3):1077—1095, 2024. doi:10.2298/CSIS230709028S.

[13] William Steingartner and Davorka Radaković. Semantic Frameworks for Selected Domain-Specific Languages in Our Research: Methodologies and Applications. In Paula Miranda and Pedro Isaías, editors, *International Conferences on Applied Computing 2024 and WWW/Internet 2024*, pages 75–82, Zagreb, Croatia, September 26–28, 2024. IADIS Press. https://www.iadisportal.org/ac-icwi-2024-proceedings.

[14] ISO: the International Organization for Standardization. ISO/IEC 9075-1:2023 Information technology — Database languages SQL — Part 1: Framework (SQL/Framework), June 2023. https://www.iso.org/standard/76583.html.

[15] Markus Voelter. *DSL Engineering — Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. https://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf.

A The Formalization of SubSQL

In this appendix, we show the manually crafted formalization of SQL, its abstract syntax (by a context free grammar), its type system (by a logical inference system), and its semantics (by a set of denotation functions that map syntactic domains to semantic ones). This syntax of the presentation is essentially the usual mathematical/logical notation, but in the form of linear (Unicode) text. In fact, our presentation just lists those files that we wrote with a simple text editor in the process of developing the formalization.

A.1 Abstract Syntax

```
SubSQL: a SQL subset
====================

Session = Stats;

Stats = Stat | Stat Stats
Stat = CreateTableStat | DropTableStat
  | InsertStat | UpdateStat | DeleteStat
  | SelectStat

CreateTableStat = "CREATE" "TABLE" ID "(" ColumnDecls ")" ";"
ColumnDecls = ColumnDecl | ColumnDecl "," ColumnDecls
ColumnDecl = ID ColumnTypeExp Constraints
ColumnTypeExp = "INT" "(" NUM ")" | "VARCHAR" "(" NUM ")"
Constraints = _ | Constraint Constraints
Constraint = "DEFAULT" Literal | "NOT NULL" | "UNIQUE" | "PRIMARY KEY"

DropTableStat = "DROP" "TABLE" ID ";"

InsertStat = "INSERT" "INTO" ID InsertColumnClause TableExp ";"
InsertColumnClause = _ | "(" IDs ")"
IDs = ID | ID "," IDs

UpdateStat = "UPDATE" ID "SET" Assignments WhereClause ";"
Assignments = Assignment | Assignment "," Assignments
Assignment = ID "=" Exp

DeleteStat = "DELETE" "FROM" ID WhereClause ";"

SelectStat = TableExp ";"

TableExp := ValuesExp | SelectExp |
  "(" TableExp ")" SetOp DistinctClause "(" TableExp ")"
SetOp = "UNION" | "INTERSECT" | "EXCEPT"
DistinctClause := _ | "ALL" | "DISTINCT"

ValuesExp = "VALUES" RowExps
RowExps = RowExp | RowExp "," RowExps
RowExp = "(" Exps ")"

SelectExp :=
```

```
  "SELECT" DistinctClause SelectClause FromClause WhereClause
   GroupByClause HavingClause OrderByClause

FromClause = _ | "FROM" Table
Table := ID AsClause | "(" TableExp ")" AsClause
  | Table JoinClause Table JoinCondition
  | Table CrossJoinClause Table
AsClause := _ | ID | "AS" ID
JoinCondition = "ON" Exp | "USING" "(" IDs ")"
JoinClause = "JOIN" | "INNER" "JOIN" |
  "LEFT" "JOIN" | "LEFT" "OUTER" "JOIN" |
  "RIGHT" JOIN" | "RIGHT" "OUTER" "JOIN" |
  "FULL" "JOIN" | "FULL" "OUTER" "JOIN"
CrossJoinClause = "CROSS" "JOIN" | ","

WhereClause := _ | "WHERE" Exp
GroupByClause := _ | "GROUP" "BY" ColumnRefs
HavingClause = _ | "HAVING" Exp

SelectClause := ColumnExps
ColumnExps := ColumnExp | ColumnExp "," ColumnExps
ColumnExp := "*" | ID "." "*" | Exp AsClause

OrderByClause := _ | "ORDER" "BY" OrderByColumns
OrderByColumns := OrderByColumn | OrderByColumn "," OrderByColumns
OrderByColumn := ColumnRef OrderClause
OrderClause := _ | "ASC" | "DESC"

Exps := Exp | Exp "," Exps
Exp := "{" ID "}" // meta-variable
  | Literal
  | ColumnRef | UnaryOp Exp | Exp BinaryOp Exp
  | "(" Exps ")"
  | "(" SelectExp ")"
  | Exp NotClause "BETWEEN" Exp "AND" Exp
  | Exp "IS" NotClause "NULL"
  | Exp "IS" NotClause "TRUE" | Exp "IS" NotClause "FALSE"
  | Exp "IS" NotClause "DISTINCT" "FROM" Exp
  | Exp "LIKE" STR
  | "CASE" WhenClauses ElseClause "END"
  | "CASE" Exp WhenClauses ElseClause "END"
  | "NULLIF" "(" Exp "," Exp ")"
  | "COALESCE" "(" Exps ")"
  | "EXISTS" "(" SelectExp ")"
  | Exp NotClause "IN" "(" Exps ")"
  | Exp NotClause "IN" "(" SelectExp ")"
  | Exp BinaryOp Quantifier "(" SelectExp ")"
  | AggregateExp // not in WHERE, only in HAVING

ColumnRef := ID | ID "." ID
ColumnRefs := ColumnRef | ColumnRef "," ColumnRefs

Literal := NUM | STR | "NULL" | "TRUE" | "FALSE"
UnaryOp := "NOT" | "-"
```

```
BinaryOp := "+" | "-" | "*" | "/" | "%"
  | "&" | "|" | "^" |
  | "=" | ">" | ">=" | "<" | "<=" | "<>"
  | "AND" | "OR"
NotClause := _ | "NOT"
Quantifier := "ANY" | "SOME" | "ALL"

WhenClauses := WhenClause | WhenClause "," WhenClauses
WhenClause := "WHEN" Exp "THEN" Exp
ElseClause = _ | "ELSE" Exp

AggregateExp := AggregateFun "(" AggregateArg ")" | "COUNT" "(" "*" ")"
AggregateArg := DistinctClause Exp
AggregateFun := "MIN" | "MAX" | "SUM" | "AVG" | "COUNT"
```

A.2 Type System

```
Type System
===========

================================================================================

Domains

================================================================================

Env :=
  // all the tables in the database
  db: TableEnv
  // the variables in the store
  × venv:VarEnv
  // the currently visible tables
  × tenv:TableEnv
  // the columns stack (including the "unnamed" ones, top is "current" table)
  × cstack:ColumnSymbols*
  // the mapping of identifiers to columns (excluding the "unnamed" ones)
  × cenv:ColumnEnv
  // determines whether symbol denotes a single cell (looked up in the
  // current row) or a full column (looked up in the current table)
  × ckind:ColumnSymbol → Kind
  // the stack position for the symbol lookup (0: bottom of the stack)
  × cdepth:ColumnSymbol → ℕ
  // the current nesting depth (length of the stack)
  × depth:ℕ
  // the set of grouped symbols
  × gsyms: ColumnSymbolSet

// the symbols in the current table
csyms: Env → ColumnSymbols
csyms(env) = env.cstack[env.depth];

VarEnv = Id →⊥ VarSymbol
VarSymbol = id:Id × vtype:VarType
```

```
VarType = int + bool + string

TableEnv = Id →⊥ TableSymbol
TableSymbol = id:Id × csyms:ColumnSymbols

ColumnEnv = Id →⊥ ColumnSymbol
ColumnSymbol = id:Id × type:ColumnType × cons:ConstraintSet × topt:IdOption
ColumnSymbols = ColumnSymbol*
ColumnSymbolSet = Set(ColumnSymbol)
ColumnType = null + int:ℕ + varchar:ℕ
ConstraintSymbol = default:Value + notnull + unique
ConstraintSet = Set(ConstraintSymbol)

Kind = column + cell
Kinds = Kind*

IdSeq = Id*
IdSet = Set(Id)
IdOption = none + id:Id

ExpType = type:Type + kind:Kind + group:Bool
ExpTypes = ExpType*

Type = null + int + bool + string + seq:Types
Types = Type*

Value = null + int:ℤ + bool:Bool + string:String + seq:Values
Values = Value*

================================================================================

Operations

================================================================================

setDBEnv: Env → Env // set database view to table environment
setDBEnv(env) ≔ env with .db ≔ env.tenv

clearTableEnv: Env → Env // set table environment to empty
clearTableEnv(env) ≔ env with .tenv ≔ ∅

setTableEnv: Env → Env // set table environment to database view
setTableEnv(env) ≔ env with .tenv ≔ env.db

toId: ID -> Id
toInt: NUM -> Int

cenv: ColumnSymbol* → ColumnEnv
cenv(csyms) ≔ λid∈Id. such csym∈range(csyms).
  csym.id = id ∧
  ¬∃csym2∈range(csyms). csym ≠ csym1 ∧ csym2.id = id;

combineSeq[A,B]: (A→B) × (A→B) → (A→B)
combineSeq(f1,f2) ≔
```

```
  let as1 = domain(f1) in
  let as2 = domain(f2) in
  let as = as1∪as2 in
  λa∈as. if a ∈ as2 then f2(a) else f1(a)

combinePar[A,B]: (A→B) × (A→B) → (A→B)
combinePar(f1,f2) :=
  let as1 = domain(f1) in
  let as2 = domain(f2) in
  let as = (as1∪as2)\(as1∩as2) in
  λa∈as. if a ∈ as2 then f2(a) else f1(a)

combineColumns: ColumnEnv × ColumnEnv → ColumnEnv
combineColumns(cenv1,cenv2) := combinePar[Id,COlumnSymbol](env1,env2)

enterTable: Env × ColumnEnv × ColumnSymbols → Env
enterTable(env,cenv,csyms) :=
  let ckind = λcsym∈range(cenv). Kind.column in
  let cdepth = λcsym∈range(cenv). env.depth in
  env
    with .cstack := cstack∘[csyms]
    with .cenv  := combineSeq[Id,ColumnSymbol](env.cenv,cenv)
    with .ckind := combineSeq[ColumnSymbol,Kind](env.ckind,ckind)
    with .cdepth = combineSeq[ColumnSymbol,ℕ](env.cdepth,cdepth)
    with .depth := env.depth+1

enterTable: Env × ColumnSymbols → Env
enterTable(env,csyms) := enterTable(env,cenv(csyms),csyms)

enterTable: Env → Env
enterTable(env) :=
  let ckind = λcsym∈range(csyms(env)). Kind.column in
  env with .ckind = combineSeq[ColumnSymbol,Kind](env.ckind,ckind)

enterRow: Env → Env
enterRow(env) :=
  let cenv = cenv(csyms(env)) in
  let ckind = λcsym∈range(cenv). Kind.cell in
  env with .ckind := combineSeq[ColumnSymbol,Kind](env.ckind,ckind)

addColumn: Env × Id × ColumnSymbol → Env
addColumn(env0,id,csym) :=
  env0 with .cenv = env0.cenv[id↦csym]

removeColumn: Env × Id → Env
removeColumn(env0,id):= env0 with .cenv :=
  env0.cenv\{⟨id,csym⟩ | csym:ColumnSymbol}

addTable: Env × Id × TableSymbol → Env
addTable(env0,id,tsym): =
  env0 with .tenv := env0.tenv[id↦tsym]

removeTable: Env × Id → Env
removeTable(env0,id):= env0 with .tenv :=
```

```
  env0.tenv\{⟨id,tsym⟩ | tsym:TableSymbol}

setGrouped: Env × ColumnSymbolSet → Env
setGrouped(env,gsyms) = env with .gsyms = gsyms

columnPositions ⊆ ColumnSymbols × ColumnSymbols × ℕ*
columnPositions(csyms0,csyms1,pos) ⇔
  domain(pos) = domain(csyms1) ∧ range(pos) = domain(csyms0) ∧
  ∀i∈domain(pos). csyms1[i] = csyms0[pos[i]]

type: ColumnType → Type
type(ctype) =
  match ctype with
    ColumnType.int(n) → Type.int
  | ColumnType.varchar(n) -> Type.string

expType: Type × ExpType × ExpType → ExpType
expType(type,etype1,etype2) →
  let kind =
    match etype1.kind,etype2.kind with
      Kind.cell,Kind.cell -> Kind.cell
      _ → Kind.column
  let grouped = etype1.grouped ∧ etype2.grouped
  in ⟨type:type,kind:kind,grouped:grouped⟩

columnsMatch ⊆ ColumnSymbol* × ColumnSymbol*
columnsMatch(csyms1,csyms2) ⇔
  domain(csyms1) = domain(csyms2) ∧
  ∀i∈domain(csyms1). columnMatch(csyms1[i],csyms2[i])

columnMatch ⊆ ColumSymbol × ColumnSymbol
columnMatch(csym1,csym2) ⇔
  (csym1.id = toId("_") ∨ csym2.id = toId("_") ∨ csym1.id = csym2.id) ∧
  columnTypeMatch(csym1.type,csym2.type)

columnTypeMatch ⊆ ColumnType × ColumnType
columnTypeMatch(ctype1,ctype2) ⇔
  (∃n1∈ℕ,n2∈ℕ. ctype1=int(n1) ∧ ctype2=int(n2)) ∨
  (∃n1∈ℕ,n2∈ℕ. ctype1=varchar(n1) ∧ ctype2=varchar(n2))

elemType: Type → Type
elemType(type) =
  match type of
    table([type0]) → type0
  | _ → type

matchExpType ⊆ ExpType × Type
matchExpType(etype,type) ⇔
  let type0 = elemType(etype.type) in
  type = Type.null ∨ type0 = Type.null ∨ type0 = type

atomicType ⊆ ExpType
atomicType(etype) ⇔
  let type0 = elemType(etype.type) in
```

```
  ¬∃types:Types. type∅ = Type.seq(types)

matchingAtomicTypes ⊆ ExpTypes × Type
matchingAtomicTypes(etypes,type) ⇔
  (type = Type.null ∨ type = Type.int ∨ Type = Type.string) ∧
  (∀i∈domain(etypes). matchExpType(etypes[i], type)

matchingAtomicTypes ⊆ ExpTypes
matchingAtomicTypes(etypes) ⇔ ∃type:Type. matchingAtomicType(etypes,type)

matchTypeColumnType ⊆ Type × ColumnType
matchTypeColumnType(type,ctype) ⇔
  case type,ctype of
    Type.null,_ → true
    Type.int,ColumnType.int(n) → true
    Type.string,ColumnType.varchar(n) → true
    _ → false

matchExpColumn ⊆ ExpType × ColumnSymbol
matchExpColumn(etype,csym) = matchTypeColumnType(etype.type,csym.type)

mergeColumns ⊆ ColumnSymbol* × ColumnSymbol* × ColumnSymbol*
mergeColumns(csyms1,csyms2,csyms) ⇔
 columnsMatch(csym1,csyms2) ∧ domain(csyms) = domain(csyms1) ∧
 ∀i∈domain(csyms). mergeColumn(csyms1[i],csyms2[i],csyms[i])

mergeColumn ⊆ ColumnSymbol × ColumnSymbol × ColumnSymbol
mergeColumn(csym1,csym2,csym) ⇔
    mergeId(csym1.id,csym2.id,csym.id) ∧
    mergeColumnType(csym1.type,csym2.type,csym.type) ∧
    csym.cons = ∅ ∧ csym.topt = IdOption.none

mergeId ⊆ Id × Id × Id
mergeId(id1,id2,id) ⇔
  if id1 = toId("_") then id = id2 else id = id1

mergeColumnType ⊆ Type × Type × Type
mergeColumnType(ctype1,ctype2,ctype) ⇔
  if ctype1 = ColumnType.null then ctype = ctype2 else ctype = ctype1

expTypesColumns ⊆ ExpTypes × ColumnSymbols
expTypesColumns(etypes,csyms) ⇔
  domain(csyms) = domain(etypes) ∧
  ∀i∈domain(csyms). expTypeColumn(etypes[i],csyms[i]⟩

expTypeColumn ⊆ ExpType × ColumnSymbol
expTypeColumn(etype,csym) ⇔
  csym.id = toId("_") ∧ expTypeColumnType(etype,ctype) ∧
  csym.cons = ∅ ∧ csym.topt = IdOption.none

expTypeColumnType ⊆ ExpType × ColumnType
expTypeColumnType(etype,ctype) ⇔
  ctype =
  case type.type of
```

```
    Type.null → ColumnType.null
  | Type.int → ColumnType.int(10)
  | Type.string → ColumnType.varchar(10)

getTypes ⊆ ColumnSymbols × Types
getTypes(csyms,types) ⇔
  domain(types) = domain(csyms) ∧
  ∀i∈domain(types). types[i] = type(csyms[i].type)

getTypes ⊆ ExpTypes × Types
getTypes(etypes,types) ⇔
  domain(types) = domain(etypes) ∧
  ∀i∈domain(types). types[i] = etypes[i].type

getKinds ⊆ ExpTypes × Kinds
getKinds(etypes,kinds) ⇔
  domain(kinds) = domain(etypes) ∧
  ∀i∈domain(kinds). kinds[i] = etypes[i].kind

getGrouped ⊆ ExpTypes × Bool*
getKinds(etypes,groupeds) ⇔
  domain(groupeds) = domain(etypes) ∧
  ∀i∈domain(groupeds). grouped[i] = etypes[i].grouped

combineKinds ⊆ Kinds × Kind
combineKinds(kinds,kind) ⇔
  if ∃i∈domain(kinds). kinds[i] = Kind.column
    then kind = Kind.column
    else kind = Kind.cell

combineGrouped ⊆ Bool* × Bool
combineKinds(groupeds,groupeds) ⇔
  if ∃i∈domain(groupeds). ¬grouped[i]
    then grouped = false
    else grouped = true

================================================================================

Judgments

================================================================================

Env ⊢ Session: session(Env)
Env ⊢ Decls: decls(Env)
Env ⊢ Decl: decl(Env)
⊢ VarTypeExp: varTypeExp(VarType)

Env ⊢ Stats: stats(Env)
Env ⊢ Stat: stat(Env)

Env ⊢ CreateTableStat: createTableStat(Env)
Id,ColumnSymbols ⊢ ColumnDecls: columnDecls(ColumnSymbols)
Id,ColumnSymbols ⊢ ColumnDecl: columnDecl(ColumnSymbols)
⊢ ColumnTypeExp: columnType(ColumnType)
```

```
ConstraintSet,ColumnType ⊢ Constraints: constraints(ConstraintSet)
ConstraintSet,ColumnType ⊢ Constraint: constraint(ConstraintSet)

Env ⊢ DropTableStat: dropTableStat(Env)

Env ⊢ InsertStat: insertStat
Env ⊢ InsertColumnClause: insertColumnClause(ColumnSymbols)
IdSeq ⊢ IDs: ids(IdSeq)

Env ⊢ UpdateStat: updateStat
Env,ColumnSymbols,IdSet ⊢ Assignments: assignments(IdSet)
Env,ColumnSymbols,IdSet ⊢ Assignment: assignment(IdSet)

Env ⊢ DeleteStat: deleteStat

Env ⊢ SelectStat: selectStat
Env ⊢ TableExp: tableExp(ColumnSymbols)
⊢ SetOp: setop(SetOpSymbol)
⊢ DistinctClause: distinctClause(DistinctSymbol)
Env ⊢ ValuesExp: valuesExp(ColumnSymbols)
Env ⊢ RowExps: rowExps(ColumnSymbols)
Env ⊢ RowExp: rowExp(ColumnSymbols)

Env ⊢ SelectExp: selectExp(ColumnSymbols)

Env ⊢ FromClause: fromClause(Env)
Env,Env,ColumnEnv,IdSet ⊢ Table: table(Env,ColumEnv,IdSet,ColumnSymbols)
⊢ AsClause: asClause(IdOption)
Env,ColumnSymbols,ColumSymbols ⊢ JoinCondition: joinCondition

Env ⊢ WhereClause: whereClause
Env ⊢ GroupByClause: groupByClause(Env)
Env ⊢ HavingClause: havingClause

Env ⊢ SelectClause: selectClause(Env,ColumnSymbols)
Env,Env,IdSet,ColumnSymbols ⊢
  ColumnExps: columnExps(Env,IdSet,ColumnSymbols,Bool)
Env,Env,IdSet,ColumnSymbols ⊢
  ColumnExp: columnExp(Env,IdSet,ColumnSymbols,Bool)

Env ⊢ OrderByClause: orderByClause
Env,ColumnSymbolSet,ColumnSymbolSet ⊢
  OrderByColumns:orderByColumns(ColumnSymbolSet)
Env,ColumnSymbolSet,ColumnSymbolSet ⊢
  OrderByColumn:orderByColumn(ColumnSymbolSet)
⊢ OrderClause: orderClause(Bool)

Env,ExpTypes ⊢ Exps: exps(ExpTypes)
Env ⊢ Exp: exp(ExpType)

Env ⊢ ColumnRef: columnRef(ColumnSymbol)
Env,ColumnSymbols ⊢ ColumnRefs: columnRefs(ColumnSymbols)

⊢ Literal: literal(Type,Value)
```

```
ExpType ⊢ UnaryOp: unaryOp(ExpType)
ExpType,ExpType ⊢ BinaryOp: binaryOp(ExpType)

Env,ExpType ⊢ WhenClauses: whenClauses(ExpType)
Env,ExpType ⊢ WhenClause: whenClause(ExpType)
Env,ExpType,ExpType ⊢ WhenClauses: whenClausesExp(ExpType)
Env,ExpType,ExpType ⊢ WhenClause: whenClauseExps(ExpType)
Env,ExpType ⊢ ElseClause: elseClause(ExpType)

Env ⊢ AggregateExp: aggregateExp(Type)
Env ⊢ AggregateArg: aggregateArg(Type)
Env,Type ⊢ AggregateFun: aggregateFun(Type)


Type System
===========

================================================================================

Inference Rules

================================================================================

Env ⊢ Session: session(Env)
===========================

env0 ⊢ Stats: stats(env)
------------------------
env0 ⊢ Stats: session(env)

Env ⊢ Stats: stats(Env)
=======================

env0 ⊢ Stat: stat(env1)
env = setDBEnv(env1)
----------------------
env0 ⊢ Stat: stats(env)

env0 ⊢ Stat: stat(env1)
env2 = setDBEnv(env1)
env2 ⊢ Stats: stats(env)
-----------------------------------------------
env0 ⊢ Stat Stats: stats(env)

Env ⊢ Stat: stat(Env)
=====================

env0 ⊢ CreateTableStat: createTableStat(env)
--------------------------------------------
env0 ⊢ CreateTableStat: stat(env)

env0 ⊢ DropTableStat: dropTableStat(env)
----------------------------------------
env0 ⊢ DropTableStat: stat(env)
```

```
env0 ⊢ InsertStat: insertStat  env = env0
---------------------------------------
env0 ⊢ InsertStat: stat(env)

env0 ⊢ UpdateStat: updateStat  env = env0
---------------------------------------
env0 ⊢ UpdateStat: stat(env)

env0 ⊢ DeleteStat: deleteStat  env = env0
---------------------------------------
env0 ⊢ DeleteStat: stat(env)

env1 = clearTableEnv(env0)
env1 ⊢ SelectStat: selectStat  env = env0
---------------------------------------
env0 ⊢ SelectStat: stat(env)

Env ⊢ CreateTableStat: createTableStat(Env)
=========================================

id = toId(ID)  id ∉ domain(env0.tenv)  id,[] ⊢ ColumnDecls: columnDecls(csyms)
tsym = ⟨id:id,csyms:csyms⟩
env = env0 with .tenv := env.tenv[id↦tsym]
-------------------------------------------------------------------------
env0 ⊢ "CREATE" "TABLE" ID "(" ColumnDecls ")" ";": createTableStat(env)

Id,ColumnSymbols ⊢ ColumnDecls: columnDecls(ColumnSymbols)
=========================================================

id,csyms0 ⊢ ColumnDecl: columnDecl(csyms)
-----------------------------------------
id,csyms0 ⊢ ColumnDecl: columnDecls(csyms)

id,csyms0 ⊢ ColumnDecl: columnDecl(csyms1)
id,csyms1 ⊢ ColumnDecls0: columnDecls(csyms)
-------------------------------------------------------
id,csyms0 ⊢ ColumnDecl "," ColumnDecls: columnDecls(csyms)

Id,ColumnSymbols ⊢ ColumnDecl: columnDecl(ColumnSymbols)
=======================================================

cid = toId(ID)  ¬∃i∈domain(csyms0). csyms0[i].id = cid
⊢ ColumnTypeExp: columnTypeExp(ctype)  ∅,ctype ⊢ Constraints: constraints(cset)
csym = ⟨id:id,type:ctype,cons:cset,topt:id(tid)⟩  csyms = csyms0∘[csym]
-------------------------------------------------------------
tid,csyms0 ⊢ ID ColumnTypeExp Constraints: columnDecl(csyms)

⊢ ColumnTypeExp: columnTypeExp(ColumnType)
==========================================

n = toInt(NUM)  type = ColumnType.int(n)
----------------------------------------
⊢ "INT" "(" NUM ")": columnType(ctype)
```

```
n = toInt(NUM)  type = ColumnType.varchar(n)
-------------------------------------------
⊢ "VARCHAR" "(" NUM ")": columnType(ctype)

ConstraintSet,ColumnType ⊢ Constraints: constraints(ConstraintSet)
==================================================================

cset = cset0
--------------------------------
cset0,ctype ⊢ _: constraints(cset)

cset0,ctype ⊢ Constraint: constraint(cset1)
cset1,ctype ⊢ Constraints: constraints(cset)
-----------------------------------------------------
cset0,ctype ⊢ Constraint Constraints: constraints(cset)

ConstraintSet,ColumnType ⊢ Constraint: constraint(ConstraintSet)
================================================================

¬∃value∈Value. Constraint.default(value) ∈ cset0
⊢ Literal: literal(type,value)
case ctype of int(n) → type = Type.int | varchar(n) → type = Type.string
cset = cset0 ∪ { Constraint.default(value) }
-------------------------------------------------
cset0,ctype ⊢ "DEFAULT" Literal: constraint(cset)

Constraint.notnull ∉ cset0
cset = cset0 ∪ { Constraint.notnull }
----------------------------------------
cset0,ctype ⊢ "NOT NULL": constraint(cset)

Constraint.unique ∉ cset0
cset = cset0 ∪ { Constraint.unique }
---------------------------------------
cset0,ctype ⊢ "UNIQUE": constraint(cset)

Constraint.notnull ∉ cset0  Constraint.unique ∉ cset0
cset = cset0 ∪ { Constraint.notnull, Constraint.unique }
-------------------------------------------
cset0,ctype ⊢ "PRIMARY KEY": constraint(cset)

Env ⊢ DropTableStat: dropTableStat(Env)
=======================================

id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv
env = env0 with .tenv := env0.tenv\{⟨id,tsym⟩}
-------------------------------------------------
env0 ⊢ "DROP" "TABLE" ID ";": dropTableStat(env)

Env ⊢ InsertStat: insertStat
============================

id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv  env1 = enterTable(env,tsym.csyms)
```

```
env1 ⊢ InsertColumnClause: insertColumnClause(csyms1)
env1 ⊢ TableExp: tableExp(csyms2)
columnsMatch(csyms1,csyms2)
-------------------------------------------------------------------------
env0 ⊢
  "INSERT" "INTO" ID^tsym InsertColumnClause^csyms1 TableExp ";": insertStat

Env ⊢ InsertColumnClause: insertColumnClause(ColumnSymbols)
=========================================================

csyms = csyms(env)
---------------------------------
env ⊢ _: insertColumnClause(csyms)

[] ⊢ IDs: ids(ids)
¬∃i1∈domain(ids),i2∈domain(ids): i1 ≠ i2 ∧ ids[i1] = ids[i2]
domain(csyms) = domain(ids) ∧ ∀i∈domain(ids). ⟨ids[i],csyms[i]⟩ ∈ env.cenv
-------------------------------------------------------------------------
env ⊢ "(" IDs ")": insertColumnClause(csyms)

IdSeq ⊢ Ids: ids(IdSeq)
=======================

ids = ids0
------------------
ids0 ⊢ _: ids(ids)

id = toId(ID)  ids1 = ids0∘[id] ids1 ⊢ IDs: ids(ids)
---------------------------------------------------
ids0 ⊢ ID "," IDs: ids(ids)

Env ⊢ UpdateStat: updateStat
============================

id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv  env1 = enterTable(env,tsym.csyms)
env1 ⊢ WhereClause: whereClause
env2 = enterRow(env1)
env2,tsym.csyms,∅ ⊢ Assignments: assignments(ids)
-------------------------------------------------------------------
env ⊢ "UPDATE" ID^tsym "SET" Assignments WhereClause ";": updateStat

Env,ColumnSymbols,IdSet ⊢ Assignments: assignments(IdSet)
=========================================================

env,csyms,ids0 ⊢ Assignment: assignment(ids)
-------------------------------------------
env,csyms,ids0 ⊢ Assignment: assignments(ids)

env,csyms,ids0 ⊢ Assignment: assignment(ids1)
env,csyms,ids1 ⊢ Assignment: assignments(ids)
-------------------------------------------------------------
env,csyms,ids0 ⊢ Assignment "," Assignments: assignments(ids)

Env,ColumnSymbols,IdSet ⊢ Assignment: assignment(Id)
```

```
=================================================

id = toId(id)  id ∉ids0  ids = ids0∪{id}
⟨id,csym⟩ ∈ env.cenv  cpos ∈ ℕ  csyms[cpos]=csym
env ⊢ Exp: exp(etype)
matchExpColumn(etype, csym)
-------------------------------------------------
env,csyms,ids0 ⊢ Id "="^cpos Exp: assignment(ids)

Env ⊢ DeleteStat: deleteStat
============================

id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv  env1 = enterTable(env,tsym.csyms)
env1 ⊢ WhereClause: whereClause
-----------------------------------------------------
env: "DELETE" "FROM" ID^tsym WhereClause ";": deleteStat

Env ⊢ SelectStat: selectStat
============================

env ⊢ TableExp: tableExp(csyms)
------------------------------
env ⊢ TableExp ";": selectStat

Env ⊢ TableExp: tableExp(ColumnSymbols)
=======================================

env ⊢ ValuesExp: valuesExp(csyms)
---------------------------------
env ⊢ ValuesExp: tableExp(csyms)

env ⊢ SelectExp: selectExp(csyms)
---------------------------------
env ⊢ SelectExp: tableExp(csyms)

env ⊢ TableExp1: tableExp(csyms1)
env ⊢ TableExp2: tableExp(csyms2)
mergeColumns(csyms1,csyms2,csyms)
-----------------------------------------------------------------
env ⊢  TableExp1 SetOp DistinctClause TableExp2: tableExp(csyms)

Env ⊢ ValuesExp: valuesExp(ColumnSymbols)
=========================================

env ⊢ RowExps: rowExps(csyms)
-------------------------------------
env ⊢ "VALUES" RowExps: valuesExp(csyms)

Env ⊢ RowExps: rowExps(ColumnSymbols)
=====================================

env ⊢ RowExp: rowExp(csyms)
---------------------------
env ⊢ RowExp: rowExps(csyms)
```

```
env ⊢ rowExp: rowExp(csyms1)
env ⊢ rowExp: rowExp(csyms2)
mergeColumns(csyms1,csyms2,csyms)
---------------------------------------
env ⊢ RowExp "," RowExps: rowExps(csyms)

Env ⊢ RowExp: rowExp(ColumnSymbols)
===================================

env,[] ⊢ Exps: exps(etypes)
expTypesColumns(etypes,csyms)
--------------------------------
env ⊢ "(" Exps "): rowExp(csyms)

Env ⊢ SelectExp: selectExp(ColumnSymbols)
=========================================

env0 ⊢ FromClause: fromClause(env1)
env1 ⊢ WhereClause: whereClause
env1 ⊢ GroupByClause: groupByClause(env2)
env2 ⊢ HavingClause: havingClause
env2 ⊢ SelectClause: selectClause(env3,csyms)
env3 ⊢ orderByClause
-------------------------------------------------------------------
env0 ⊢
  "SELECT" DistinctClause SelectClause FromClause WhereClause
   GroupByClause HavingClause OrderByClause ";": selectExp(csyms)

Env ⊢ FromClause: fromClause(Env)
=================================

env = enterTable(env0,[])
---------------------------------
env0 ⊢ _: fromClause(env)

cenv0 = env0.cenv
setTableEnv(env0),env0,cenv0,∅ ⊢ Table: table(env1,cenv1,ids,csyms)
env = enterTable(env1,cenv1,csyms)
-----------------------------------------------------------------------------
env0 ⊢ "FROM" Table: fromClause(env)

Env,Env,ColumnEnv,IdSet ⊢ Table: table(Env,ColumnEnv,IdSet,ColumnSymbols)
=========================================================================

id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv  csyms = tsym.csyms
⊢ AsClause: asClause(idopt)
case idopt of
  Id.none → ids = ids0∪{id} ∧
    env = if id ∈ ids0 removeTable(env1,id) else addTable(env1,id,tsym) ∧
    cenv = combineColumns(cenv1,cenv(csyms))
| Id.some(id0) → id0 ∉ ids0 ∧ ids = ids0∪{id,id0} ∧
    env = addTable(env1,id0,tsym) ∧ cenv = cenv1
-------------------------------------------------------------
```

```
env0,env1,cenv1,ids0 ⊢ ID^tsym AsClause: table(env,cenv,ids,csyms)

env0 ⊢ TableExp: tableExp(csyms)
⊢ AsClause: asClause(idopt)
case idopt of
  Id.none → ids = ids0 ∧ env = env1 ∧ cenv = combineColumns(cenv1,cenv(csyms))
| Id.some(id0) -> id0 ∉ ids0 ∧ ids = ids0∪{id0} ∧
    tsym = ⟨id0,csyms:csyms⟩ ∧ env = addTable(env1,id0,tsym) ∧ cenv = cenv1
-------------------------------------------------------------------------------
env0,env1,cenv1,ids0 ⊢ "(" TableExp ")" AsClause: table(env,cenv,ids,csyms)

env0,env1,cenv1,ids0 ⊢ Table1: table(env2,cenv2,ids1,csyms1)
env0,env2,cenv2,ids1 ⊢ Table2: table(env,cenv,ids,csyms2)
env,csyms1,csyms2 ⊢ JoinCondition: joinCondition
csyms = csyms1∘csyms2
-----------------------------------------------------------------
env0,env1,cenv1,ids0 ⊢
  Table1 JoinClause Table2 JoinCondition: table(env,cenv,ids,csyms)

env0,env1,cenv1,ids0 ⊢ Table1: table(env2,cenv2,ids1,csyms1)
env0,env2,cenv2,ids1 ⊢ Table2: table(env,cenv,ids,csyms2)
csyms = csyms1∘csyms2
-----------------------------------------------------------
env0,env1,cenv1,ids0 ⊢
  Table1 CrossJoinClause Table2: table(env,cenv,ids,csyms)

⊢ AsClause: asClause(IdOption)
==============================

idOption = IdOption.none
------------------------
⊢ _: asClause(idOption)

id = toId(ID)  idOption = IdOption.id(id)
-----------------------------------------
⊢ ID: asClause(idOption)

id = toId(ID)  idOption = IdOption.id(id)
-----------------------------------------
⊢ "AS" ID: asClause(idOption)

Env,ColumnSymbols,ColumSymbols ⊢ JoinCondition: joinCondition
============================================================

env1 = enterRow(enterTable(env,csyms1∘csyms2))
env1 ⊢ Exp:exp(etype)
etype.type = Type.bool
------------------------------------------
env,csyms1,csyms2 ⊢ "ON" Exp: joinCondition

[] ⊢ IDs: ids(ids)
n1 = csyms1.length
∀i∈domains(ids). ∃j∈domains(csyms1). ids[i] = csyms1[j].id
∀i∈domains(ids). ∃j∈domains(csyms2). ids[i] = csyms2[j].id
```

```
pos1 = λi∈domain(ids). choose j∈domains(csyms1). ids[i] = csyms1[j].id
pos2 = λi∈domain(ids). n1 + choose j∈domains(csyms2). ids[i] = csyms2[j].id
-----------------------------------------------------------
env,csyms1,csyms2 ⊢ "USING" "(" IDs^(pos1,pos2) ")": joinCondition

Env ⊢ WhereClause: whereClause
==============================

(axiom, no prerequisites)
-------------------------
env ⊢ _: whereClause

env0 = enterRow(env)
env0 ⊢ Exp: exp(etype)
matchExpType(etype, Type.bool)
--------------------------------------
env ⊢ "WHERE" Exp: whereClause

Env ⊢ GroupByClause: groupByClause(Env)
=======================================

(axiom, no prerequistes)
env0 = env
-----------------------------
env ⊢ _: groupByClause(env0)

env,[] ⊢ ColumnRefs: columnRefs(csyms)
pos∈ℕ*  columnPositions(csyms(env),csyms,pos)
env0 = setGrouped(env,csyms)
-----------------------------------------------------------
env ⊢ "GROUP" "BY" ColumnRefs^pos: groupByClause(env0)

Env ⊢ HavingClause: havingClause
================================

(axiom, no prerequisites)
-------------------------
env ⊢ _: havingClause

env0 = enterRow(env)
env0 ⊢ Exp: exp(etype)
matchExpType(etype, Type.bool)
etype.kind = Kind.cell
etype.grouped = true
--------------------------------
env ⊢ "HAVING" Exp: havingClause

Env ⊢ SelectClause: selectClause(Env,ColumnSymbols)
===================================================

env1 = env0 with .cenv = ∅
env0,env1,∅,[] ⊢ ColumnExps: columnExps(env2,ids,csyms,grouped)
env = enterTable(env2,csyms)
-----------------------------------------
```

```
env0 ⊢ ColumnExps^grouped: selectClause(env,csyms)

Env,Env,IdSet,ColumnSymbols ⊢
  ColumnExps: columnExps(Env,IdSet,ColumnSymbols,Bool)
=============================================================================

env0,env1,idset0,csyms0 ⊢ ColumnExp: columnExp(env,idset,csyms,grouped)
-------------------------------------------------------------
env0,env1,idset0,csyms0 ⊢ ColumnExp: columnExps(env,idset,csyms,grouped)

env0,env1,idset0,csyms0 ⊢ ColumnExp: columnExp(env2,idset1,csyms1,grouped1)
env0,env2,idset1,csyms1 ⊢ ColumnExps: columnExps(env,idset,csyms,grouped2)
grouped = grouped1 ∧ grouped2
-----------------------------------------------------------------------------
env0,env1,idset0,csyms0 ⊢ ColumnExp "," ColumExps:
  columnExps(env,idset,csyms,grouped)

Env,IdSet,ColumnSymbols ⊢ ColumnExp: columnExp(Env,IdSet,ColumnSymbols,Bool)
=============================================================================

csyms1 = csyms(env0)  csyms = csyms0∘csyms1
env = env1 with .cenv = cenv(csyms)
idset = idset0 ∪ { csym.id | csym ∈ range(csyms1) }
grouped ⇔ range(csyms1) ⊆ range(env.gsyms)
-----------------------------------------------------------------------
env0,env1,idset0,csyms0 ⊢ "*": columnExp(env,idset,csyms,grouped)

id = toId(ID)  ⟨id,tsym⟩ ∈ env0.tenv
csyms1 = tsym.csyms  csyms = csyms0∘csyms1
env = env1 with .cenv = cenv(csyms)
idset = idset0 ∪ { csym.id | csym ∈ range(csyms1) }
pos∈ℕ*  columnPositions(csyms(env0),csyms1,pos)
grouped ⇔ range(csyms1) ⊆ range(env.gsyms)
-----------------------------------------------------------------
env0,env1,idset0,csyms0 ⊢ ID "." "*"^pos: columnExp(env,idset,csyms,grouped)

env2 = enterRow(env0)
env2 ⊢ Exp: exp(etype)
⊢ AsClause: asClause(idopt)
expTypeColumnType(etype,ctype)
case idopt of
  Id.some(id0) →
    id = id0 ∧ id0 ∉ idset0 ∧
    csym = ⟨id:id,type:ctype,cons:∅,tid:none⟩
| Id.none →
    case Exp of
      ColumnRef(ref) →
        env0 ⊢ ref: columnRef(csym)
        id = csym.id
   | default →
        id = toId("_")
        csym = ⟨id:id,type:ctype,cons:∅,tid:none⟩
idset = idset0∪{id}
env = if id ∈ idset0 then removeColumn(env1,id) else addColumn(env1,id,csym)
```

```
csyms = csyms0∘[csym]
if ∃cref:ColumnRef,csym,pos,kind,depth. Exp=cref^{csym,pos,kind,depth} then
  ∃cref:ColumnRef,csym,pos,kind,depth. Exp=cref^{csym,pos,kind,depth} ∧
    grouped ⇔ csym in range(env.gsyms)
else
  grouped ⇔ etype.grouped
-------------------------------------------------------------------------
env0,env1,idset0,csyms0 ⊢ Exp AsClause: columnExp(env,idset,csyms,grouped)

Env ⊢ OrderByClause: orderByClause
==================================

(axiom, no prerequisites)
---------------------------
env ⊢ _: orderByClause

env,∅ ⊢ OrderByColumns: orderByColumns^{pos,asc}(csymset)
-----------------------------------------------------------------
env ⊢ "ORDER" "BY" OrderByColumns^{pos,asc}: orderByClause

Env,ColumnSymbolSet ⊢ OrderByColumns:orderByColumns(ColumnSymbolSet)
=====================================================================

env,csymset0 ⊢ OrderByColumn^{p,a}: orderByColumn(csymset)
pos = [p]  asc = [a]
-----------------------------------------------------------
env,csymset0 ⊢ OrderByColumn^{pos,asc}: orderByColumns(csymset)

env,csymset0 ⊢ OrderByColumn^{p,a}: orderByColumn(csymset1)
env,csymset1 ⊢ OrderByColumns^{pos0,asc0}: orderByColumns(csymset)
pos = [p]∘pos0  asc = [a]∘asc0
-----------------------------------------------------------------
env,csymset0 ⊢ (OrderByColumn "," OrderByColumns)^{pos,asc}:
  orderByColumns(csymset)

Env,ColumnSymbolSet ⊢ OrderByColumn:orderByColumn(ColumnSymbolSet)
=====================================================================

env ⊢ ColumnRef: columnRef(csym)
csym ∉ csymset0
csymset = csymset0∪{csym}
columnPositions(csyms(env),[csym],[p])
⊢ OrderClause: orderClause(a)
-------------------------------------------------------------------
env,csymset0 ⊢ (ColumnRef OrderClause)^{p,a}: orderByColumn(csymset)

⊢ OrderClause(Bool)
====================

a = true
--------------------
⊢ _: orderClause(a)

a = true
```

```
-----------------------
⊢ "ASC": orderClause(a)

a = false
-----------------------
⊢ "DESC: orderClause(a)

Env,ExpTypes ⊢ Exps: exps(ExpTypes)
=========================================

env ⊢ Exp: exp(etype)
etypes = etypes0∘[etype]
------------------------------
env,etypes0 ⊢ Exp: exps(etypes)

env ⊢ Exp: exp(etype)
env,etypes0∘[etype] ⊢ Exps: exps(etypes)
--------------------------------------
env,etypes0 ⊢ Exp "," Exps: exps(etypes)

Env ⊢ Exp: exp(ExpType)
=======================

id = toId(ID)  ⟨id,vtype⟩ ∈ env.venv
case vtype of
  VarType.int → type = Type.int
| VarType.bool → type = Type.bool
| VarType.string → type = string
| _ → false
etype = ⟨type:type,kind:Kind.cell,grouped:true⟩
-----------------------------------------------
env ⊢ {" ID "}": exp(etype)

⊢ Literal: literal(type,value)
etype = ⟨type:type,kind:Kind.cell,grouped:true⟩
-----------------------------------------------
env ⊢ Literal^value: exp(etype)

env ⊢ ColumnRef: columnRef(csym)
type = type(csym.type)
kind = env.ckind(csym)
depth = env.cdepth(csym)
pos ∈ ℕ  env.cstack[depth][pos] = csym
grouped = depth < length(env.cstack)-1 ∨ csym ∈ env.gsyms
etype = ⟨type:type, kind:kind, group:grouped⟩
-------------------------------------------------------
env ⊢ ColumnRef^{csym,pos,kind,depth}: exp(etype)

env ⊢ Exp: exp(etype0)
etype0 ⊢ UnaryOp: unaryOp(etype)
-----------------------------------
env ⊢ UnaryOp Exp: exp(etype)

env ⊢ Exp1: exp(etype1)
```

```
env ⊢ Exp2: exp(etype2)
etype1,etype2 ⊢ BinaryOp: binaryOp(etype)
-----------------------------------------
env ⊢ Exp1 BinaryOp Exp2: exp(etype)

env,[] ⊢ Exps: exps(etypes)
if length(etypes) = 1 then
  etype = etypes[0]
else
  getTypes(etypes,types) ∧
  getKinds(etypes,kinds) ∧
  getGrouped(etypes,groupeds) ∧
  combineKinds(kinds,kind) ∧
  combineGrouped(groupeds,grouped) ∧
  etype = ⟨type:Type.seq(types),kind:kind,grouped:grouped⟩
-----------------------------------------------------------
env ⊢ "(" Exps ")": exp(etype)

env ⊢ SelectExp: selectExp(csyms)
getTypes(csyms,types)
if length(types) = 1 then
  etype = ⟨type:types[0],kind:Kind.column,grouped:false⟩
else
  etype = <type:seq(types),kind:Kind.column,grouped:false⟩
-----------------------------------------------------------
env ⊢ "(" SelectExp ")": exp(etype)

env ⊢ Exp1: exp(etype1)
env ⊢ Exp2: exp(etype2)
env ⊢ Exp3: exp(etype3)
matchingAtomicTypes([etype1,etype2,etype3])
combineKinds([etype1.kind,etype2.kind,etype3.kind],kind)
combineGrouped([etype1.grouped,etype2.grouped,etype3.grouped],grouped)
etype = ⟨type:Type.bool,kind:kind,grouped:grouped⟩
-------------------------------------------------------------------
env ⊢ Exp1 NotClause "BETWEEN" Exp2 "AND" Exp3: exp(etype)

env ⊢ Exp: exp(etype0)
atomicExpType(etype0);
etype = ⟨type:Type.bool,kind:etype0.kind,grouped:etype0.grouped⟩
-----------------------------------------------------------------
env ⊢ Exp "IS" NotClause "NULL": exp(etype)

env ⊢ Exp: exp(etype0)
matchExpType(etype0,Type.bool)
etype = ⟨type:Type.bool,kind:etype0.kind,grouped:etype0.grouped⟩
-----------------------------------------------------------------
env ⊢ Exp "IS" NotClause "TRUE": exp(etype)

env ⊢ Exp: exp(etype0)
matchExpType(etype0,Type.bool)
etype = ⟨type:Type.bool,kind:etype0.kind,grouped:etype0.grouped⟩
-----------------------------------------------------------------
env ⊢ Exp "IS" NotClause "FALSE": exp(etype)
```

```
env ⊢ Exp1: exp(etype1)
env ⊢ Exp2: exp(etype2)
matchingAtomicTypes([etype1,etype2])
combineKinds([etype1.kind,etype2.kind],kind)
combineGroups([etype1.grouped,etype2.grouped],grouped])
etype = ⟨type:Type.bool,kind:kind,grouped:grouped⟩
-------------------------------------------------------------
env ⊢ Exp1 "IS" NotClause "DISTINCT" "FROM" Exp2: exp(etype)

env ⊢ Exp: exp(etype0)
matchExpType(etype0,Type.string)
etype = ⟨type:Type.bool,kind:etype0.kind,grouped:etype0.grouped⟩
-----------------------------------------------------------------
env ⊢ Exp "LIKE" STR: exp(etype)

etype0 = ⟨type:Type.null,kind:Kind.cell,grouped:true⟩
env,etype0 ⊢ WhenClauses: whenClauses(etype1)
env,etype1 ⊢ ElseClause: elseClause(etype)
-----------------------------------------------------
env ⊢ "CASE" WhenClauses ElseClause "END": exp(etype)

etype0 = ⟨type:Type.null,kind:Kind.cell,grouped:true⟩
env ⊢ Exp: exp(etype1)
env,etype0,etype1 ⊢ WhenClauses: whenClausesExp(etype2)
env,etype2 ⊢ ElseClause: elseClause(etype)
-------------------------------------------------------------
env ⊢ "CASE" Exp WhenClauses ElseClause "END": exp(etype)

env ⊢ Exp1: exp(etype1)
env ⊢ Exp2: exp(etype2)
matchingAtomicTypes([etype1,etype2])
combineKinds([etype1.kind,etype2.kind],kind)
combineGrouped([etype1.grouped,etype2.grouped],grouped)
etype = ⟨type:etype1.type,kind:kind,grouped:grouped⟩
-----------------------------------------------------
env ⊢ "NULLIF" "(" Exp1 "," Exp2 ")": exp(etype)

env ⊢ Exps: exps(etypes)
matchingAtomicTypes(etypes,type)
getKinds(etypes,kinds)
getGroupeds(etypes,groupeds)
combineKinds(kinds,kind)
combineGrouped(groupeds,grouped)
etype = ⟨type:type,kind:kind,grouped:grouped⟩
-------------------------------------------
env ⊢ "COALESCE" "(" Exps ")": exp(etype)

env ⊢ SelectExp: selectExp(csyms)
etype = ⟨type:Type.bool,kind:Kind.cell,grouped:true⟩
-----------------------------------------------
env ⊢ "EXISTS" "(" SelectExp ")": exp(etype)

env ⊢ Exp: exp(etype0)
```

```
env ⊢ Exps: exps(etypes)
matchingAtomicTypes([etype0]∘etypes)
getKinds(etypes,kinds)
getGroupeds(etypes,groupeds)
combineKinds([etype0.kind]∘kinds,kind)
combineGroupeds([etype0.grouped]∘groupeds,grouped)
etype = ⟨type:Type.bool,kind:kind,grouped:grouped⟩
-------------------------------------------------
env ⊢ Exp NotClause "IN" "(" Exps ")": exp(etype)

env ⊢ Exp: exp(etype0)
env ⊢ SelectExp: selectExp(csyms)
length(csyms) = 1
type1 = etype0.type
type2 = type(csyms[0].type)
matchingAtomicTypes([type1,type2])
etype = ⟨type:Type.bool,kind:etype0.kind,grouped:etype0.grouped⟩
-----------------------------------------------------------------
env ⊢ Exp NotClause "IN" "(" SelectExp ")": exp(etype)

env ⊢ Exp: exp(etype0)
env ⊢ SelectExp: selectExp(csyms)
length(csyms) = 1
etype1 = ⟨type:type(csyms[0].type),kind:Kind.cell,grouped:true⟩
etype0,etype1 ⊢ BinaryOp: binaryOp(etype)
-----------------------------------------------------------------
env ⊢ Exp BinaryOp Quantifier "(" SelectExp ")": exp(etype)

env ⊢ AggregateExp: aggregateExp(type)
etype = ⟨type:type,kind:Kind.cell,grouped:true⟩
-------------------------------------------------
env ⊢ AggregateExp: aggregateExp(etype)

Env ⊢ ColumnRef: columnRef(ColumnSymbol)
=========================================

id = toId(ID)
⟨id,csym⟩ ∈ env.cenv
-----------------------------------------
env ⊢ ID: columnRef(csym)

⟨id,csym⟩ in env.cenv
tid = toId(ID1)  cid = toId(ID2)
⟨tid,tsym⟩ ∈ env.tenv  <cid,csym> ∈ cenv(tsym.csyms)
-----------------------------------------------------
env ⊢ ID1 "." ID2: columnRef(csym)

Env,ColumnSymbols ⊢ ColumnRefs: columnRefs(ColumnSymbols)
=========================================================

env ⊢ ColumnRef: columnRef(csym)
csyms = csyms0∘[csym]
-----------------------------------------
env,csyms0 ⊢ ColumnRef: columnRefs(csyms)
```

```
env ⊢ ColumnRef: columnRef(csym)
csyms1 = csyms0 ∘[csym]
env,csyms1 ⊢ ColumnRefs: columnRefs(csyms)
-----------------------------------------------------
env,csyms0 ⊢ ColumnRef "," ColumnRefs: columnRefs(csyms)

⊢ Literal: literal(Type,Value)
==============================

type = Type.int
value = Value.int(toInt(NUM))
-----------------------------
⊢ NUM: literal(type,value)

type = Type.string
value = Value.string(STR)
-------------------------
⊢ STR: literal(type,value)

type = Type.null
value = Value.null
--------------------------
⊢ NULL: literal(type,value)

type = Type.bool
value = Value.bool(true)
----------------------------
⊢ "TRUE": literal(type,value)

type = Type.bool
value = Value.bool(false)
-----------------------------
⊢ "FALSE": literal(type,value)

ExpType ⊢ UnaryOp: unaryOp(ExpType)
===================================

matchExpType(etype0,Type.bool)
etype = ⟨type:Type.bool,kind:etype0.kind,grouped:etype0.grouped⟩
-----------------------------------------------------------------
etype0 ⊢ NOT: unaryOp(etype)

matchExpType(etype0,Type.int)
etype = ⟨type:Type.int,kind:etype0.kind,grouped:etype0.grouped⟩
----------------------------------------------------------------
etype0 ⊢ "-": unaryOp(etype)

ExpType,ExpType ⊢ BinaryOp: binaryOp(ExpType)
=============================================

matchExpType(etype1,Type.int)  matchExpType(etype2,Type.int)
etype = expType(Type.int,etype1,etype2)
---------------------------------------------------------
```

```
etype1,etype2 ⊢ "+": binaryOp(etype)

matchExpType(etype1,Type.int)  matchExpType(etype2,Type.int)
etype = expType(Type.int,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "-": binaryOp(etype)

matchExpType(etype1,Type.int)  matchExpType(etype2,Type.int)
etype = expType(Type.int,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "*": binaryOp(etype)

matchExpType(etype1,Type.int)  matchExpType(etype2,Type.int)
etype = expType(Type.int,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "/": binaryOp(etype)

matchExpType(etype1,Type.int)  matchExpType(etype2,Type.int)
etype = expType(Type.int,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "%": binaryOp(etype)

matchExpType(etype1,Type.int)  matchExpType(etype2,Type.int)
etype = expType(Type.int,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "&": binaryOp(etype)

matchExpType(etype1,Type.int)  matchExpType(etype2,Type.int)
etype = expType(Type.int,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "|": binaryOp(etype)

matchExpType(etype1,Type.int)  matchExpType(etype2,Type.int)
etype = expType(Type.int,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "^": binaryOp(etype)

matchingAtomicTypes([etype1,etype2])
etype = expType(Type.bool,etype1,etype2)
----------------------------------------
etype1,etype2 ⊢ "=": binaryOp(etype)

matchingAtomicTypes([etype1,etype2])
etype = expType(Type.bool,etype1,etype2)
----------------------------------------
etype1,etype2 ⊢ "<": binaryOp(etype)

matchingAtomicTypes([etype1,etype2])
etype = expType(Type.bool,etype1,etype2)
----------------------------------------
etype1,etype2 ⊢ "<=": binaryOp(etype)

matchingAtomicTypes([etype1,etype2])
etype = expType(Type.bool,etype1,etype2)
```

```
---------------------------------------
etype1,etype2 ⊢ ">": binaryOp(etype)

matchingAtomicTypes([etype1,etype2])
etype = expType(Type.bool,etype1,etype2)
---------------------------------------
etype1,etype2 ⊢ ">=": binaryOp(etype)

matchingAtomicTypes([etype1,etype2])
etype = expType(Type.bool,etype1,etype2)
---------------------------------------
etype1,etype2 ⊢ "<>": binaryOp(etype)

matchExpType(etype1,Type.bool)  matchExpType(etype2,Type.bool)
etype = expType(Type.bool,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "AND": binaryOp(etype)

matchExpType(etype1,Type.bool)  matchExpType(etype2,Type.bool)
etype = expType(Type.bool,etype1,etype2)
-------------------------------------------------------------
etype1,etype2 ⊢ "OR": binaryOp(etype)

Env,ExpType ⊢ WhenClauses: whenClauses(ExpType)
===============================================

env,etype0 ⊢ WhenClause: whenClause(etype)
------------------------------------------
env,etype0 ⊢ WhenClause: whenClauses(etype)

env,etype0 ⊢ WhenClause: whenClause(etype1)
env,etype1 ⊢ WhenClauses: whenClauses(etype)
-------------------------------------------------------------
env,etype0 ⊢ WhenClause "," WhenClauses: whenClauses(etype)

Env,ExpType ⊢ WhenClause: whenClause(ExpType)
=============================================

env ⊢ Exp1: exp(etype1)
env ⊢ Exp2: exp(etype2)
matchesExpType(etype1,Type.bool)
matchingAtomicTypes([etype0,etype2],type)
combineKinds([etype0.kind,etype1.kind,etype2.kind],kind)
combineGrouped([etype0.grouped,etype1.grouped,etype2.grouped],grouped)
etype = ⟨type:type,Kind:kind,grouped:grouped⟩
-----------------------------------------------------
env,etype0 ⊢ "WHEN" Exp1 "THEN" Exp2: whenClause(etype)

Env,ExpType,ExpType ⊢ WhenClauses: whenClausesExp(ExpType)
==========================================================

env,etype0,etype3 ⊢ WhenClause: whenClauseExp(etype)
----------------------------------------------------
env,etype0,etype3 ⊢ WhenClause: whenClausesExp(etype)
```

```
env,etype0,etype3 ⊢ WhenClause: whenClauseExp(etype1)
env,etype1,etype3 ⊢ WhenClauses: whenClausesExp(etype)
-------------------------------------------------------------------
env,etype0,etype3 ⊢ WhenClause "," WhenClauses: whenClausesExp(etype)

Env,ExpType,ExpType ⊢ WhenClause: whenClauseExp(ExpType)
=======================================================

env ⊢ Exp1: exp(etype1)
env ⊢ Exp2: exp(etype2)
matchingAtomicTypes(etype3,etype1)
matchingAtomicTypes([etype0,etype2],type)
combineKinds([etype0.kind,etype1.kind,etype2.kind],kind)
combineGrouped([etype0.grouped,etype1.grouped,etype2.grouped],grouped)
etype = ⟨type:type,Kind:kind,grouped:grouped⟩
-------------------------------------------------------------------
env,etype0,etype3 ⊢ "WHEN" Exp1 "THEN" Exp2: whenClauseExp(etype)

Env,ExpType ⊢ ElseClause: elseClause(ExpType)
=============================================

etype = etype0
----------------------------------
env,etype0 ⊢ "_": elseClause(etype)

env ⊢ Exp: exp(etype1)
matchingAtomicTypes([etype0,etype1],type)
combineKinds([etype0.kind,etype1.kind],kind)
combineGrouped([etype0.grouped,etype1.grouped],grouped)
etype = ⟨type:type,Kind:kind,grouped:grouped⟩
-------------------------------------------
env,etype0 ⊢ "ELSE" Exp: elseClause(etype)

Env ⊢ AggregateExp: aggregateExp(Type)
======================================

env0 = enterTable(env)
env0 ⊢ AggregateArg: aggregateArg(etype)
etype.type ⊢ AggregateFun: aggregateFun(type)
-------------------------------------------------------------
env ⊢ AggregateFun "(" AggregateArg ")": aggregateExp(type)

type = Type.int
-----------------------------------------------
env ⊢ "COUNT" "(" "*" ")" : aggregateExp(type)

Env ⊢ AggregateArg: aggregateArg(ExpType)
=========================================

env ⊢ Exp: exp(etype)
etype.kind = Kind.column
etype.grouped = false
-----------------------------------------------
```

```
env ⊢ DistinctClause Exp: aggregateArg(etype)

Type ⊢ AggregateFun: aggregateFun(Type)
=====================================

type0 = Type.int ∨ type = Type.string
type = type0
---------------------------------
type0 ⊢ "MIN": aggregateFun(type)

type0 = Type.int ∨ type = Type.string
type = type0
---------------------------------
type0 ⊢ "MAX": aggregateFun(type)

type0 = Type.int
type = type0
---------------------------------
type0 ⊢ "SUM": aggregateFun(type)

type0 = Type.int
type = type0
---------------------------------
type0 ⊢ "AVG": aggregateFun(type)

type = Type.int
-----------------------------------
type0 ⊢ "COUNT": aggregateFun(type)
```

A.3 Denotational Semantics

```
Denotational Semantics
======================

================================================================================

Semantic Domains

================================================================================

State = store:Store × db:Database × tables:Tables × rows:Table

Store = Id →⊥ Value
Database := Id →⊥ Table

Row = Value*
Table = Row*
Tables = Table*

ValueOption = none + some:Value

================================================================================
```

```
Semantic Operations

================================================================================

top[T]: T* → T
top[T](seq) := let n = length(seq) in seq[n-1]

newTable: State × Table → State
newTable(s,table) := s with .tables := s.tables∘[table];

currentTable: State → Table
currentTable(s) := top[Table](s.tables)

newRow: State × Row → Row
newRow(s,row) := s with .rows := s.rows∘[row];

currentRow: State → Row
currentRow(s) := top[Row](s.rows)

tableRows: TableSymbol × ColumnSymbols → (Table → Table)
tableRows^tsym_csyms(rows) :=
  λi∈domain(rows). tableRow^tsym_csyms(rows[i])

tableRow: TableSymbol × ColumnSymbols → (Row → Row)
tableRow^tsym_csyms(row) :=
  λi∈domain(tsym.csyms).
    let tcsym = tsym.csyms[i] in
    such value∈Value.
      if ∃i0∈domain(csyms). csyms[i0] = tcsym then
        ∃i0∈domain(csyms). csyms[i0] = tcsym ∧ value = row[i0]
      else if ∃v0∈Value. ConstraintSymbol.default(v0) ∈ tcsym.cons then
        ConstraintSymbol.default(value) ∈ tcsym.cons
      else
        value = Value.null

satisfiedConstraints ⊆ TableSymbol × Table
satisfiedConstraints(tsym,table) ⇔
  ∀i∈domain(tsym.csyms).
    let cset = tsym.csyms(i).cons in
    (ConstraintSymbol.notnull ∈ cset ⇒
      ∀j∈domain(table). table[j][i] ≠ Value.null) ∧
    (ConstraintSymbol.unique ∈ cset ⇒
      ∀j1∈domain(table),j2∈domain(table).
        j1 ≠ j2 ⇒ table[j1][i] ≠ table[j2][i])

select: Table × Set(Nat) → Table
select(rows,is) :=
  let is0 = { i ∈ Nat | i < |is|} in
  λi∈is0.
   choose i0∈is with i = |{i1 ∈ is | i1 < i0 }|.
   rows[i0]

filter: Table × Pred(Row) → Table
filter(table,pred) :=
```

```
  let is = { i∈domain(table) | pred(table[i]) } in
  select(table, is)

concatenate: Tables → Table
concatenate(tables) :=
  let n = length(tables) in
  let is = domain(tables)∪{n} in
  let ns = λi∈is. sum_j∈is with j < i. length(tables[i]) in
  let is0 = { i∈ℕ | i < ns[n] } in
  λi0∈is0. let i = max_i∈is with i0 ≥ ns[i] in tables[i][i0-ns[i]]

crossJoin: Table × Table → Table
crossJoin(table1,table2) :=
  let n1 = length(table1) in
  let n2 = length(table2) in
  let is = { i∈Nat | i < n1*n2 } in
  λi∈is.
    let i1 = is/n2 in
    let i2 = is%n2 in
    table1[i1]∘table2[i2]

null: ℕ → Values
nulls(n) := choose seq∈Values. length(seq) = n ∧
  ∀i∈domain(seq). seq[i] = Value.null

allRows: Table → Table
allRows(table) := table

distinctRows: Table → Table
distinctRows(table) :=
  let is1 = domain(table) in
  let is2 = { i1∈is1 | ¬∃i2∈is1. i2 < i1 ∧ table[i1] = table[i2] } in
  select(table,is2)

isTrue ⊆ Value
isTrue(value) ⇔ value = Value.bool(true)

project: Row × ℕ* → Values
project(row,pos) := λi∈domain(pos). row[pos[i]]

enumeration[T]: Set(T) → T*
enumeration[T](set) := choose seq∈T*. enumeration(set,seq)

enumeration[T] ⊆ Set(T) × T*
enumeration[T](set,seq) :=
  |seq| = |set| ∧
  ∀i∈domain(seq). seq[i] ∈ set ∧ ¬∃j∈domain(seq). j ≠ i ∧ seq[j] = seq[i]

permutation ⊆ Table × Table
permutation(table1,table2) ⇔
  domain(table1) = domain(table2) ∧
  ∀row∈Row. |{ i∈Domain(table1). table1[i] = row }|
          = |{ i∈Domain(table2). table2[i] = row }|
```

```
sorted ⊆ Table × ℕ* × Bool*
sorted(table, pos, asc) ⇔
  ∀i∈domain(table),j∈domain(table). i < j ⇒
    if asc
      then lesseq(table[i],table[j],pos,0)
      else lesseq(table[j],table[i],pos,0)

lesseq ⊆ Row × Row × Nat* × Nat
lesseq(row1,row2,pos,i) ⇔
  i = |domain(pos)| ∨ less(row1[i],row2[i]) ∨
      (row1[i] = row2[i] ∧ lesseq(row1,row2,pos,i+1))

less ⊆ Value × Value
less(v1,v2) ⇔
  match v1,v2 with
    _,Value.null → false
  | Value.null,_ → true
  | Value.int(i1),Value.int(i2) → i1 < i2
  | Value.string(s1),Value.string(v2) → s1 "before" s2

like ⊆ Str × Str
like(s1,s2) ≔ ... // semantics of "LIKE"

isIn: Value × Values → Value
isIn(v,vs) ⇔
  if v = Value.null then
    Value.null
  else if ∃i∈domain(vs). vs[i] = v then
    Value.bool(true)
  else if ∃i∈domain(vs). vs[i] = Value.null
    Value.null
  else
    Value.bool(false)

intFun: Value × Value × (ℤ×ℤ→ℤ) → Value
intFun(v1,v2,fun) =
  match v1,v2 with
    Value.null,_ → Value.null
  | _,Value.null → Value.null
  | Value.int(i1),Value.int(i2) → Value.int(fun(i1,i2))

intFun: Value × Value × Pred(Value) → Value
valuePred(v1,v2,pred) =
  match v1,v2 with
    Value.null,_ → Value.bool(false)
  | _,Value.null → Value.bool(false)
  | _,_ → if pred(v1,v2) then Value.bool(true) else Value.bool(false)

================================================================================

Denotations

================================================================================
```

```
⟦Session⟧: State →⊥ State × Tables
⟦Stats⟧: State →⊥ State × Tables
⟦Stat⟧: State →⊥ State × Table

⟦CreateTableStat⟧: State →⊥ State
⟦DropTableStat⟧: State →⊥ State

⟦InsertStat⟧: State →⊥ State
⟦Assignments⟧: State × Row → Row
⟦Assignment⟧: State × Row → Row

⟦UpdateStat⟧: State →⊥ State
⟦DeleteStat⟧: State →⊥ State

⟦SelectStat⟧: State →⊥ Table

⟦SelectStat⟧: State →⊥ Table
⟦TableExp⟧: State →⊥ Table
⟦SetOp⟧: Table × Table → Table
⟦DistinctClause⟧: Table × (Table → Table) → Table

⟦ValuesExp⟧: State →⊥ Table
⟦RowsExp⟧: State →⊥ Table
⟦RowExp⟧: State →⊥ Row

⟦SelectExp⟧: State →⊥ Table

⟦FromClause⟧: State →⊥ Table
⟦Table⟧: State →⊥ Table
⟦JoinClause⟧: Table × Table →⊥ Table
⟦JoinCondition⟧: State → Pred(Row)

⟦WhereClause⟧: State × Table → Table
⟦WhereClause⟧: State → Pred(Row)
⟦GroupClause⟧: Table → Tables
⟦HavingClause⟧: State × Tables → Tables

⟦SelectClause⟧: State × Tables →⊥ Table
⟦ColumnExps⟧: State →⊥ Table
⟦ColumnExp⟧: State →⊥ Table

⟦OrderByClause⟧: Table → Table

⟦Exps⟧: State →⊥ Values
⟦Exp⟧: State →⊥ Value

⟦UnaryOp⟧: State × (State →⊥ Value) →⊥ Value
⟦BinaryOp⟧: State × (State →⊥ Value) × (State →⊥ Value) →⊥ Value
⟦NotClause⟧: Value → Value
⟦Quantifier⟧: Values × (Value → Value) →⊥ Value

⟦WhenClauses⟧: State →⊥ ValueOption
⟦WhenClauses⟧: State × Value →⊥ ValueOption
⟦WhenClause⟧: State →⊥ ValueOption
```

```
⟦WhenClause⟧: State × Value →⊥ ValueOption
⟦ElseClause⟧: State →⊥ Value

⟦AggregateExp⟧: State →⊥ Value
⟦AggregateArg⟧: State →⊥ Values
⟦AggregateFun⟧: Values →⊥ Value



Denotational Semantics
======================

================================================================================

Equations

================================================================================

⟦Session⟧: State →⊥ State × Tables
==================================

⟦Stats⟧(s) := ⟦Stats⟧(s)

⟦Stats⟧: State →⊥ State × Tables
================================

⟦Stat⟧(s) := ⟦Stat⟧(s)
⟦Stat Stats⟧(s) :=
  let ⟨s1,tables1⟩ = ⟦Stat⟧(s) in
  let ⟨s2,tables2⟩ = ⟦Stats⟧(s1) in
  ⟨s2,tables1∘tables2⟩

⟦Stat⟧: State →⊥ State × Tables
==============================

⟦CreateTableStat⟧(s) = ⟨⟦CreateTableStat⟧(s),[]⟩
⟦DropTableStat⟧(s) = ⟨⟦DropTableStat⟧(s),[]⟩
⟦InsertStat⟧(s) = ⟨⟦InsertStat⟧(s),[]⟩
⟦UpdateStat⟧(s) = ⟨⟦UpdateStat⟧(s),[]⟩
⟦DeleteStat⟧(s) = ⟨⟦DeleteStat⟧(s),[]⟩
⟦SelectStat⟧(s) = ⟨s,[ ⟦SelectStat⟧(s) ]⟩

⟦CreateTableStat⟧: State →⊥ State
=================================

⟦"CREATE" "TABLE" ID "(" ColumnDecls ")" ";"⟧(s) :=
  s WITH .db = s.db[ toId(ID) ↦ [] ]

⟦DropTableStat⟧: State →⊥ State
===============================

⟦"DROP" "TABLE" ID ";"⟧(s) := s WITH .db =
  let id = toId(ID) in s with .db = s.db\{ ⟨id,t⟩ | t ∈ Table }
```

```
⟦InsertStat⟧: State →⊥ State
============================

⟦"INSERT" "INTO" ID^tsym InsertColumnClause^csyms TableExp ";"⟧(s) :=
  let table = s.db[tsym.id] in
  let s0 = newTable(s,table) in
  let rows1 = ⟦TableExp⟧(s0) in
  let rows2 = tableRows^tsym_csyms(rows1) in
  let rows3 = table∘rows2 in
  if ¬satisfiedConstraints(tsym,rows3))
    then ⊥
    else s with .db = s.db[tsym.id↦rows3]

⟦UpdateStat⟧: State →⊥ State
============================

⟦"UPDATE" ID^tsym "SET" Assignments WhereClause ";"⟧(s) :=
  let table = s.db[tsym.id] in
  let s0 = newTable(s,table) in
  let table0 =
    λi∈domain(table).
      let row = table[i] in
      if ¬⟦WhereClause⟧(s0)(row) then
        row
      else
        let s1 = newRow(s0,row) in
        ⟦Assignments⟧(s1,row)
   in if ¬satisfiedConstraints(tsym,table0))
    then ⊥
    else s with .db[tsym.id↦table0]

⟦Assignments⟧: State × Row → Row
================================

⟦Assignment⟧(s,row) := ⟦Assignment⟧(s,row)

⟦Assignment "," Assignments⟧(s,row) :=
  let row1 = ⟦Assignment⟧(s,row) in
  ⟦Assignments⟧(s,row1)

⟦Assignment⟧: State × Row → Row
================================

⟦Id "="^cpos Exp⟧(s,row) :=
  let value = ⟦Exp⟧(s) in row[cpos↦value]

⟦DeleteStat⟧: State →⊥ State
============================

⟦"DELETE" "FROM" ID^tsym WhereClause ";"⟧(s) :=
  let table = s.db[tsym.id] in
  let s0 = newTable(s,table) in
  let is = { i ∈ domain(table) |
    let row = table[i] in
```

```
    ¬⟦WhereClause⟧(s0)(row) } in
  let table0 = select(table,is) in
  s with .db[tsym.id↦table0]

⟦SelectStat⟧: State →⊥ Table
============================

⟦TableExp ";"⟧(s) := ⟦TableExp⟧(s)

⟦TableExp⟧: State →⊥ Table
==========================

⟦ValuesExp⟧(s) := ⟦ValuesExp⟧(s)

⟦SelectExp⟧(s) := ⟦SelectExp⟧(s)

⟦TableExp1 SetOp DistinctClause TableExp2⟧(s) :=
  let table1 = ⟦TableExp1⟧(s) in
  let table2 = ⟦TableExp2⟧(s) in
  let table3 = ⟦SetOp⟧(table1,table2) in
  ⟦DistinctClause⟧(distinctRows,table3)

⟦SetOp⟧: Table × Table → Table
==============================

⟦"UNION"⟧(table1,table2) := table1∘table2

⟦"INTERSECT"⟧(table1,table2) :=
  let is1 = domain(table1) in
  let is2 = domain(table2) in
  let is3 = { i1∈is1 | ∃i2∈is2. table1[i1] = table2[i2] } in
  select(table1,is3)

⟦"EXCEPT"⟧(table1,table2) :=
  let is1 = domain(table1) in
  let is2 = domain(table2) in
  let is3 = { i1∈is1 | ¬∃i2∈is2. table1[i1] = table2[i2] } in
  select(table1,is3)

⟦DistinctClause⟧: (Table → Table) × Table → Table
=================================================

⟦"_"⟧(default,ta) := default(ta)

⟦"ALL"⟧(default,table) := allRows(table)

⟦"DISTINCT"⟧(default,table) := distinctRows(table)

⟦ValuesExp⟧: State →⊥ Table
===========================

⟦"VALUES" RowExps⟧ := ⟦RowExps⟧

⟦RowExps⟧: State →⊥ Table
```

```
=========================

⟦RowExp⟧(s) := let row = ⟦RowExp⟧(s) in [row]

⟦RowExp "," RowExps⟧(s) :=
  let row = ⟦RowExp⟧(s) in
  let rows = ⟦RowExps⟧(s) in
  [row]∘rows

⟦RowExp⟧: State →⊥ Row
======================

⟦"(" Exps ")"⟧ := ⟦Exps⟧

⟦SelectExp⟧: State →⊥ Table
===========================

⟦"SELECT" DistinctClause SelectClause FromClause^csyms0 WhereClause
          GroupByClause HavingClause OrderByClause ";"⟧(s) :=
  let table1 = ⟦FromClause⟧(s) in
  let s0 = newTable(s,table1) in
  let table2 = ⟦WhereClause⟧(s0,table1) in
  let tables3 = ⟦GroupClause⟧(table2) in
  let tables4 = ⟦HavingClause⟧(s0,tables3)) in
  let table5 = ⟦SelectClause⟧(s0,tables4) in
  let table6 = ⟦DistinctClause⟧(allRows,table5) in
  let table7 = ⟦OrderByClause⟧(table6) in
  table7

⟦FromClause⟧: State →⊥ Table
============================

⟦_⟧(s) := []
⟦"FROM" Table⟧ := ⟦Table⟧

⟦Table⟧: State →⊥ Table
=======================

⟦ID^tsym AsClause⟧(s) := s.db[tsym.id]

⟦"(" TableExp ")" AsClause⟧(s) := ⟦TableExp⟧(s)

⟦Table1^csyms1 JoinClause Table2^csyms2 JoinCondition⟧(s) :=
  let table1 = ⟦Table1⟧(s) in
  let table2 = ⟦Table2⟧(s) in
  let n1 = length(csyms1) in
  let n2 = length(csyms2) in
  ⟦JoinClause⟧(table1,table2,n1,n2,⟦JoinCondition⟧(s))

⟦Table1 CrossJoinClause Table⟧(s) :=
  let table1 = ⟦Table1⟧(s) in
  let table2 = ⟦Table2⟧(s) in
  crossJoin(table1,table2)
```

```
⟦JoinClause⟧: Table × Table × ℕ × ℕ × Pred(Row) →⊥ Table
=======================================================

⟦"JOIN"⟧ := ⟦"INNER" "JOIN"⟧

⟦"INNER" "JOIN"⟧(table1,table2,n1,n2,pred) :=
  filter(crossJoin(table1,table2),pred)

⟦"LEFT" "JOIN"⟧ := ⟦"LEFT" "OUTER" "JOIN"⟧

⟦"LEFT" "OUTER" "JOIN"⟧(table1,table2,n1,n2,pred) :=
  let tables =
    λi∈domain(table1).
     let row1 = table1[i] in
     let jrows1 = λj∈domain(table2). row1∘table2[j] in
     let jrows2 = filter(jrows1,pred) in
     let jrows3 = if jrows2 ≠ [] then jrows2 else [ row1∘nulls(n2) ] in
     jrows3
  in concatenate(tables)

⟦"RIGHT" "JOIN"⟧ := ⟦"RIGHT" "OUTER" "JOIN"⟧

⟦"RIGHT" "OUTER" "JOIN"⟧(table1,table2,n1,n2,pred) :=
  let tables =
    λj∈domain(table2).
     let row2 = table2[j] in
     let jrows1 = λj∈domain(table1). table1[i]∘row2 in
     let jrows2 = filter(jrows1,pred) in
     let jrows3 = if jrows2 ≠ [] then jrows2 else [ nulls(n1)∘row2 ] in
     jrows3
  in concatenate(tables)

⟦"FULL" "JOIN"⟧ := ⟦"FULL" "OUTER" "JOIN"⟧

⟦"FULL" "JOIN"⟧(table1,table2,n1,n2,pred) :=
  let table = filter(crossJoin(table1,table2),pred) in
  let tables1 =
    λi∈domain(table1).
     let row1 = table1[i] in
     let jrows1 = λj∈domain(table2). row1∘table2[j] in
     let jrows2 = filter(jrows1,pred) in
     let jrows3 = if jrows2 ≠ [] then [] else [ row1∘nulls(n2) ] in
     jrows3
  let tables2 =
    λj∈domain(table2).
     let row2 = table2[j] in
     let jrows1 = λj∈domain(table1). table1[i]∘row2 in
     let jrows2 = filter(jrows1,pred) in
     let jrows3 = if jrows2 ≠ [] then jrows2 else [ nulls(n1)∘row2 ] in
     jrows3
  in table∘concatenate(tables1)∘concatenate(tables2)

⟦JoinCondition⟧: State → Pred(Row)
==================================
```

```
⟦"ON" Exp⟧(s) :=
  λrow∈Row. isTrue(⟦Exp⟧(newRow(s,row)))

⟦"USING" "(" IDs^pos1,pos2 ")"⟧(s) :=
  λrow∈Row. ∀i∈domain(pos1).
    let value1 = row[pos1[i]] in
    let value2 = row[pos2[i]] in
    value1 ≠ Value.null ∧ value2 ≠ Value.null ∧ value1 = value2

⟦WhereClause⟧: State × Table → Table
====================================
[ ⟦WhereClause⟧(s,table) = filter(table, ⟦WhereClause⟧(s)) ]

⟦_⟧(s,table) := table

⟦"WHERE" Exp⟧(s,table) :=
  filter(table, λrow∈Row. isTrue(⟦Exp⟧(newRow(s,row))))

⟦WhereClause⟧: State → Pred(Row)
================================

⟦_⟧(s) := λrow∈Row. true

⟦"WHERE" Exp⟧(s) := λrow∈Row. isTrue(⟦Exp⟧(newRow(s,row)))

⟦GroupClause⟧: Table → Tables
=============================

⟦_⟧(table) := [table]

⟦"GROUP" "BY" ColumnRefs^pos⟧(table) :=
  let keys =
    enumeration[Values]
      { key∈Values | ∃i∈domain(table). key = project(table[i],pos) }
  in λi∈domain(keys).
      enumeration[Table] { row∈range(table) | keys[i] = project(row) }

⟦HavingClause⟧: State × Tables → Tables
=======================================

⟦_⟧(s,tables) := tables

⟦"HAVING" Exp⟧(s,tables) :=
  filter(tables, λtable∈Table. isTrue(⟦Exp⟧(newRow(s,table[0]))))

⟦SelectClause⟧: State × Tables →⊥ Table
=======================================

⟦ColumnExps^grouped⟧(s,tables) :=
  let tables0 =
    λi∈domain(tables).
      let s0 = newTable(s,tables[i]) in
      let table = ⟦ColumnExps⟧(s0);
```

```
      if length(table) > 0 ∧ grouped then [table[0]] else table
  in concatenate(tables0)

⟦ColumnExps⟧: State →⊥ Table
============================

⟦ColumnExp⟧(s) := ⟦ColumnExp⟧(s) in

⟦ColumnExp "," ColumnExps⟧(s) :=
    let table1 = ⟦ColumnExp⟧(s) in
    let table2 = ⟦ColumnExps⟧(s) in
    λj∈domain(currentTable(s)). table1[j]∘table2[j]

⟦ColumnExp⟧: State →⊥ Table
===========================

⟦"*"⟧(s) := currentTable(s)

⟦ID "." "*"^pos⟧(s) :=
  let table = currentTable(s) in
  λi∈domain(table). project(table[i],pos)

⟦Exp AsClause⟧(s) :=
  let table = currentTable(s) in
  λi∈domain(table).
    let v = ⟦Exp⟧(newRow(s,table[i]))
    in match v with Value.seq(vs) → vs | _ → [v]

⟦OrderByClause⟧: Table → Table
==============================

⟦_⟧(table) := table

⟦"ORDER" "BY" OrderByColumns^{pos,asc}⟧(table) :=
  choose table0∈Table:
    permutation(table0,table) ∧
    sorted(table0,pos,asc)

⟦Exps⟧: State →⊥ Values
=======================

⟦Exp⟧(s) := [ ⟦Exp⟧(s) ]
⟦Exp "," Exps⟧(s) := [ ⟦Exp⟧(s) ] ∘ ⟦Exps⟧(s)

⟦Exp⟧: State →⊥ Value
=====================

⟦"{" ID "}"⟧(s) := s.store(toId(ID))

⟦Literal^value⟧(s) := value

⟦ColumnRef^{csym,pos,kind,depth}⟧(s) :=
  match kind with
    Kind.cell → s.rows[depth][pos]
```

```
  | Kind.column → Value.seq(λi∈domain(s.tables[depth]). s.tables[depth][i][pos])

⟦UnaryOp Exp⟧(s) := ⟦UnaryOp⟧(s,⟦Exp⟧)

⟦Exp1 BinaryOp Exp2⟧(s) := ⟦BinaryOp⟧(s,⟦Exp1⟧,⟦Exp2⟧)

⟦"(" Exps ")"⟧(s) :=
  let vs = ⟦Exps⟧(s) in
  if length(vs) = 1 then vs[0] else Value.seq(vs)

⟦"(" SelectExp ")"⟧(s) :=
  let table = ⟦SelectExp⟧(s) in
  if length(table) ≠ 1 then
    ⊥
  else
    let vs = table[0] in
    if length(vs) = 1 then vs[0] else Value.seq(vs)

⟦Exp1 NotClause "BETWEEN" Exp2 "AND" Exp3⟧(s) :=
  let v = ⟦Exp1⟧(s) in
  let e = λs∈State. v in
  let e1 = λs∈State. ⟦<=⟧(s,⟦Exp2⟧,e) in
  let e2 = λs∈State. ⟦<=⟧(s,e,⟦Exp3⟧) in
  ⟦NotClause⟧(λs∈State. ⟦"AND"⟧(s,e1,e2))

⟦Exp "IS" NotClause "NULL"⟧(s) :=
  let v = ⟦Exp⟧(s) in
  ⟦NotClause⟧(Value.bool(v = Value.null))

⟦Exp "IS" NotClause "TRUE"⟧(s) :=
  let v = ⟦Exp⟧(s) in
  ⟦NotClause⟧(if v = Value.null then v else Value.bool(v = Value.bool(true)))

⟦Exp "IS" NotClause "FALSE"⟧(s) :=
  let v = ⟦Exp⟧(s) in
  ⟦NotClause⟧(if v = Value.null then v else Value.bool(v = Value.bool(false)))

⟦Exp1 "IS" NotClause "DISTINCT" "FROM" Exp2⟧(s) :=
  let v1 = ⟦Exp1⟧(s) in
  let v2 = ⟦Exp2⟧(s) in
  ⟦NotClause⟧(Value.bool(v1 ≠ v2))

⟦Exp "LIKE" STR⟧(s) :=
  let v = ⟦Exp⟧(s) in
  match v with
    Value.null → Value.null
    Value.string(str) → Value.bool(like(str,STR))

⟦"CASE" WhenClauses ElseClause "END"⟧(s) :=
  let vo = ⟦WhenClauses⟧(s) in
  match vo with
    ValueOption.none → ⟦ElseClause⟧(s)
  | ValueOption.some(v) → v
```

```
⟦"CASE" Exp WhenClauses ElseClause "END"⟧(s) :=
  let ve = ⟦Exp⟧(s) in
  let vo = ⟦WhenClauses⟧(s,ve) in
  match vo with
    ValueOption.none → ⟦ElseClause⟧(s)
  | ValueOption.some(v) → v

⟦"NULLIF" "(" Exp1 "," Exp2 ")"⟧(s) :=
  let v1 = ⟦Exp1⟧(s) in
  let v2 = ⟦Exp2⟧(s) in
  if v1 = v2 then Value.null else v1

⟦"COALESCE" "(" Exps ")"⟧(s) :=
  let vs = ⟦Exps⟧(s) in
  if ¬∃i∈domain(vs). vs[i] ≠ Value.null then
    Value.null
  else
    let i = min i∈domain(vs). vs[i] ≠ Value.null in
    vs[i]

⟦"EXISTS" "(" SelectExp ")"⟧(s) :=
  let table = ⟦SelectExp⟧(s) in
  Value.bool(table.length > 0)

⟦Exp NotClause "IN" "(" Exps ")"⟧(s) :=
  let v = ⟦Exp⟧(s) in
  let vs = ⟦Exps⟧(s) in
  ⟦NotClause⟧(isIn(v,vs))

⟦Exp NotClause "IN" "(" SelectExp ")"⟧(s) :=
  let v = ⟦Exp⟧(s) in
  let table = ⟦SelectExp⟧(s) in
  let vs = λi∈domain(table). table[i](0) in
  ⟦NotClause⟧(isIn(v,vs))

⟦Exp BinaryOp Quantifier "(" SelectExp ")"⟧(s) :=
  let v = ⟦Exp⟧(s) in
  let table = ⟦SelectExp⟧(s) in
  let vs = λi∈domain(table). table[i](0) in
  ⟦Quantifier⟧(vs, λv0∈Value.⟦BinaryOp⟧(s,λs∈State.v,λs∈State.v0))

⟦AggregateExp⟧(s) := ⟦AggregateExp⟧(s)

⟦UnaryOp⟧: State × (State →⊥ Value) →⊥ Value
==========================================

⟦"NOT"⟧(s,e) :=
  let v = e(s) in
  match v with
    Value.bool(true) → Value.bool(false)
  | Value.bool(false) → Value.bool(true)
  | _ → Value.null

⟦"-"⟧(s,e) :=
```

```
  let v = e(s) in
  match v with
  | Value.int(i) → Value.int(-i)
  | _ → Value.null

⟦BinaryOp⟧: State × (State →⊥ Value) × (State →⊥ Value) →⊥ Value
================================================================

⟦"+"⟧(s,e1,e2) := intFun(e1(s),e2(s),λi1∈Int,i2∈Int.i1+i2)
⟦"-"⟧(s,e1,e2) := intFun(e1(s),e2(s),λi1∈Int,i2∈Int.i1-i2)
⟦"*"⟧(s,e1,e2) := intFun(e1(s),e2(s),λi1∈Int,i2∈Int.i1·i2)
⟦"/"⟧(s,e1,e2) := intFun(e1(s),e2(s),λi1∈Int,i2∈Int.i1/i2)
⟦"%"⟧(s,e1,e2) := intFun(e1(s),e2(s),λi1∈Int,i2∈Int.i1 mod i2))
⟦"&"⟧(s,e1,e2) := intFun(e1(s),e2(s),λi1∈Int,i2∈Int.i1 bitand i2)
⟦"|"⟧(s,e1,e2) := intFun(e1(s),e2(s),λi1∈Int,i2∈Int.i1 bitor i2)
⟦"^"⟧(s,e1,e2) := intFun(e1(s),e2(s),λi1∈Int,i2∈Int.i1 bitxor i2)

⟦"="⟧(s,e1,e2) := valuePred(e1(s),e2(s),λv1∈Value,i2∈Value.v1=v2)
⟦">"⟧(s,e1,e2) := valuePred(e1(s),e2(s),λv1∈Value,i2∈Value.less(v2,v1))
⟦">="⟧(s,e1,e2) := valuePred(e1(s),e2(s),λv1∈Value,i2∈Value.v1=v2∨less(v2,v1))
⟦"<"⟧(s,e1,e2) := valuePred(e1(s),e2(s),λv1∈Value,i2∈Value.less(v1,v2))
⟦"<="⟧(s,e1,e2) := valuePred(e1(s),e2(s),λv1∈Value,i2∈Value.v1=v2∨less(v1,v2))
⟦"<>"⟧(s,e1,e2) := valuePred(e1(s),e2(s),λv1∈Value,i2∈Value.v1≠v2)

⟦"AND"⟧(s,e1,e2) :=
  match s(e1) with
    Value.bool(false) → Value.bool(false)
  | Value.bool(b1) →
      match s(e2) with
        Value.bool(false) → Value.bool(false)
      | Value.bool(b2) →
          match b1,b2 with
            true,true → Value.bool(true)
          | _ → Value.null

⟦"OR"⟧(s,e1,e2) :=
  match s(e1) with
    Value.bool(true) → Value.bool(true)
  | Value.bool(b1) →
      match s(e2) with
        Value.bool(true) → Value.bool(true)
      | Value.bool(b2) →
          match b1,b2 with
            false,false → Value.bool(false)
          | _ → Value.null

⟦NotClause⟧: Value → Value
==========================

⟦_⟧(v) := v

⟦"NOT"⟧(v) :=
  match v with
    Value.bool(true) → Value.bool(false)
```

```
  | Value.bool(false) → Value.bool(true)
  | _ → Value.null

⟦Quantifier⟧: Values × (Value → Value) →⊥ Value
================================================

⟦"ANY"⟧ := ⟦"SOME"⟧

⟦"SOME"⟧(vs,p) :=
  if ∃i∈domain(vs). p(v) = Value.bool(true) then
    Value.bool(true)
  else if ∃i∈domain(vs). p(v) = Value.null then
    Value.null
  else
    Value.bool(false)

⟦"ALL"⟧(vs,p) :=
  if ∃i∈domain(vs). p(v) = Value.bool(false) then
    Value.bool(false)
  else if ∃i∈domain(vs). p(v) = Value.null then
    Value.null
  else
    Value.bool(true)

⟦WhenClauses⟧: State →⊥ ValueOption
===================================

⟦WhenClause⟧(s) := ⟦WhenClause⟧(s)

⟦WhenClause "," WhenClauses⟧(s) :=
  let vo = ⟦WhenClause⟧(s) in
  match vo with
    Value.some(v) → vo
  | _ → ⟦WhenClauses⟧(s)

⟦WhenClauses⟧: State × Value →⊥ ValueOption
===========================================

⟦WhenClause⟧(s,v) := ⟦WhenClause⟧(s,v)

⟦WhenClause "," WhenClauses⟧(s,v) :=
  let vo = ⟦WhenClause⟧(s,v) in
  match vo with
    Value.some(v) → vo
  | _ → ⟦WhenClauses⟧(s,v)

⟦WhenClause⟧: State →⊥ ValueOption
==================================

⟦"WHEN" Exp1 "THEN" Exp2⟧(s) :=
  match ⟦Exp1⟧(s) with
  | Value.bool(true) → ValueOption.some(⟦Exp2⟧(s))
  | Value.bool(false) → ValueOption.none
  | _ → ValueOption.some(Value.null)
```

```
⟦WhenClause⟧: State × Value →⊥ ValueOption
==========================================

⟦"WHEN" Exp1 "THEN" Exp2⟧(s,ve) ≔
  let ve1 = ⟦Exp1⟧(s) in
  match ve,ve1 with
    Value.null,_ → Value.null
  | _,Value.null → Value.null
  | _,_ → if ve = ve1 then ValueOption.some(⟦Exp2⟧(s)) else ValueOption.none

⟦ElseClause⟧: State →⊥ Value
============================

⟦"_"⟧(s) ≔ Value.null

⟦"ELSE" Exp⟧(s) ≔ ⟦Exp⟧(s)

⟦AggregateExp⟧: State →⊥ Value
==============================

⟦AggregateFun "(" AggregateArg ")"⟧(s) :=
  let vs = ⟦AggregateArg⟧(s) in
  ⟦AggregateFun⟧(vs)

⟦"COUNT" "(" "*" ")"⟧(s) ≔ Value.int(length(currentTable(s)))

⟦AggregateArg⟧: State →⊥ Values
===============================

⟦DistinctClause Exp⟧(s) :=
  let v = ⟦Exp⟧(s) in
  let vs = match v with Value.seq(vs) → vs | _ → [v] in
  let table1 = λi∈domain(vs). [vs(i)] in
  let table2 = ⟦DistinctClause⟧(allRows,table1)) in
  λi∈domain(table2). table2[i][0]

⟦AggregateFun⟧: Values →⊥ Value
===============================

⟦"MIN"⟧(vs) ≔
  let i = choose i∈domain(vs) with ∀j∈domain(vs). lessEqual(vs[i],vs[j])
  in vs[i]

⟦"MAX"⟧(vs) ≔
  let i = choose i∈domain(vs) with ∀j∈domain(vs). lessEqual(vs[j],vs[i])
  in vs[i]

⟦"SUM"⟧(vs) ≔ Value.int(∑i∈domain(vs). match vs[i] with Value.int(i0) → i0)

⟦"AVG"⟧(vs) ≔ ⟦"SUM"⟧(vs)/⟦"COUNT"⟧(vs)

⟦"COUNT"⟧(vs) ≔ Value.int(length(vs))
```

B The SLANG-Based Implementation of SubSQL

In this section, we give the source code of the actual implementation of SubSQL, which is derived from the language's formalization presented in Appendix A. Its core is SLANG specification file `SubSQL.txt` from which the SLANG software generates Java code for the language's abstract syntax domains, its parser, printer, type checker, and executor. This is augmented by a small set of manually crafted Java classes that implement the mathematical domains on which the type system and the denotational semantics are based. Additionally a simple database is implemented that stores its tables internally as lists of rows and externally as a text file that embeds the table data in the widely used CSV (comma separated values) format. Based on all of this, a Java class is given that implements a simple Java API by which on the one hand external SubSQL scripts can be executed and on the other hand the execution of SubSQL statements can be embedded in Java programs. The first aspect is demonstrated by a sample SubSQL script for which the output produced by the SubSQL program is shown.

B.1 SLANG Specification

```
// ----------------------------------------------------------------------
// SubSQL.txt
// A subset of SQL
// (c) 2024 https://www.risc.jku.at/research/formal/software/SLANG
// Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <william.steingartner@tuke.sk>
// ----------------------------------------------------------------------

language SubSQL
{
  target java
  {
    header
    {#
      package lang.subsql;
      import java.util.*;
      import java.util.function.*;
      import database.*;
      import static lang.subsql.SubSQL.*;
      import static subsql.TypeSystem.*;
      import static subsql.TypeSystem.check;
      import static subsql.Semantics.*;
    #}
  }

  // ----------------------------------------------------------------------
  // syntactic domains
  // ----------------------------------------------------------------------
  domains
  {
    Session = SessionStats[Stats];

    Stats = SingleStat[Stat] + MultipleStats[Stat,Stats];
```

```
Stat = StatCreateTable[CreateTableStat] + StatDropTable[DropTableStat] +
   StatInsert[InsertStat] + StatUpdate[UpdateStat] +
   StatDelete[DeleteStat] + StatSelect[SelectStat];

CreateTableStat = CreateTable[ID,ColumnDecls,#Annotation#];
ColumnDecls = SingleColumnDecl[ColumnDecl] +
  MultipleColumnDecls[ColumnDecl,ColumnDecls];
ColumnDecl = TheColumnDecl[ID,ColumnTypeExp,Constraints];
ColumnTypeExp = ColumnTypeInt[NUM] + ColumnTypeVarChar[NUM];
Constraints = NoConstraint + SomeConstraint[Constraint,Constraints];
Constraint = Default[Literal] + NotNull + Unique + PrimaryKey;

DropTableStat = DropTable[ID];

InsertStat = Insert[ID,InsertColumnClause,TableExp,#Annotation#];
InsertColumnClause = NoInsertColumnClause + SomeInsertColumnClause[IDs];
IDs = SingleID[ID] + MultipleIDs[ID,IDs];

UpdateStat = Update[ID,Assignments,WhereClause,#Annotation#];
Assignments = SingleAssignment[Assignment] +
  MultipleAssignments[Assignment,Assignments];
Assignment = TheAssignment[ID,Exp,#Annotation#];

DeleteStat = Delete[ID,WhereClause,#Annotation#];

SelectStat = Select[TableExp,#Annotation#];

TableExp = TableExpValuesExp[ValuesExp] + TableExpSelectExp[SelectExp] +
  TableExpCombination[TableExp,SetOp,DistinctClause,TableExp];
SetOp = SetUnion + SetIntersect + SetExcept;
DistinctClause = DistinctNone + DistinctAll + DistinctDistinct;

ValuesExp = TheValuesExp[RowExps];
RowExps = SingleRowExp[RowExp] + MultipleRowExps[RowExp,RowExps];
RowExp = TheRowExp[Exps];

SelectExp = TheSelectExp[DistinctClause,SelectClause,FromClause,
  WhereClause,GroupByClause,HavingClause,OrderByClause];
FromClause = NoFromClause + SomeFromClause[TableD];
TableD = TableId[ID,AsClause,#Annotation#] +
  TableTableExp[TableExp,AsClause] +
  TableJoin[TableD,JoinClause,TableD,JoinCondition,#Annotation#] +
  TableCrossJoin[TableD,CrossJoinClause,TableD];
AsClause = NoAsClause + SomeAsClause[ID] + SomeAsClauseAs[ID];
JoinClause = Join + InnerJoin + LeftJoin + LeftOuterJoin +
  RightJoin + RightOuterJoin + FullJoin + FullOuterJoin;
JoinCondition = JoinOn[Exp] + JoinUsing[IDs,#Annotation#];
CrossJoinClause = CrossJoin + CommaJoin;

WhereClause = NoWhere + SomeWhere[Exp];
GroupByClause = NoGroupBy + SomeGroupBy[ColumnRefs,#Annotation#];
HavingClause = NoHaving + SomeHaving[Exp];

SelectClause = TheSelectClause[ColumnExps,#Annotation#];
```

```
ColumnExps = SingleColumnExp[ColumnExp] +
  MultipleColumnExps[ColumnExp,ColumnExps];
ColumnExp = ColumnExpAll + ColumnExpId[ID,#Annotation#] +
  ColumnExpExp[Exp,AsClause];

OrderByClause = OrderByNone + OrderBySome[OrderByColumns,#Annotation#];
OrderByColumns = SingleOrderByColumn[OrderByColumn,#Annotation#] +
  MultipleOrderByColumns[OrderByColumn,OrderByColumns,#Annotation#];
OrderByColumn = TheOrderByColumn[ColumnRef,OrderClause,#Annotation#];
OrderClause = OrderNone + OrderAsc + OrderDesc;

Exps = SingleExp[Exp] + MultipleExps[Exp,Exps];
Exp = ExpMetaVar[ID] +
  ExpLiteral[Literal,#Annotation#] +
  ExpRef[ColumnRef,#Annotation#] +
  ExpUnary[UnaryOp,Exp] + ExpBinary[Exp,BinaryOp,Exp] +
  ExpExps[Exps] + ExpSelect[SelectExp] +
  ExpBetween[Exp,NotClause,Exp,Exp] +
  ExpIsNull[Exp,NotClause] +
  ExpIsTrue[Exp,NotClause] + ExpIsFalse[Exp,NotClause] +
  ExpIsDistinct[Exp,NotClause,Exp] +
  ExpLike[Exp,STR,#Annotation#] +
  ExpCase[WhenClauses,ElseClause] +
  ExpCaseExp[Exp,WhenClauses,ElseClause] +
  ExpNullIf[Exp,Exp] +
  ExpCoalesce[Exps] +
  ExpExists[SelectExp] +
  ExpIn[Exp,NotClause,Exps] +
  ExpInSelect[Exp,NotClause,SelectExp] +
  ExpQuantified[Exp,BinaryOp,Quantifier,SelectExp] +
  ExpAggregateExp[AggregateExp];

ColumnRef = PlainRef[ID] + TableRef[ID,ID];
ColumnRefs = SingleColumnRef[ColumnRef] +
  MultipleColumnRefs[ColumnRef,ColumnRefs];

Literal = LiteralNUM[NUM] + LiteralSTR[STR] + LiteralNull +
  LiteralTrue + LiteralFalse;
UnaryOp = UnaryNot + UnaryMinus;
BinaryOp = BinaryPlus + BinaryMinus + BinaryTimes + BinaryDiv + BinaryMod +
  BinaryBitAnd + BinaryBitOr + BinaryBitExclOr +
  BinaryEqual + BinaryGreater + BinaryGreaterEqual +
  BinaryLess + BinaryLessEqual + BinaryNotEqual +
  BinaryAnd + BinaryOr;
NotClause = NoNot + SomeNot;
Quantifier = QuantifierAny + QuantifierSome + QuantifierAll;

WhenClauses = SingleWhen[WhenClause] +
  MultipleWhens[WhenClause,WhenClauses];
WhenClause = TheWhenClause[Exp,Exp];
ElseClause = NoElse + SomeElse[Exp];

AggregateExp = AggregateExpFun[AggregateFun,AggregateArg] +
  AggregateExpCount;
```

```
  AggregateArg = TheAggregateArg[DistinctClause,Exp];
  AggregateFun = AggregateMin + AggregateMax + AggregateSum +
    AggregateAvg + AggregateCount;
}

code
{#
  // if text representation of object is not empty,
  // separate it by a space from the leading text
  public static String separate(Object object)
  {
    String text = object.toString();
    return text.equals("") ? "" : " "+text;
  }
  // if text representation of object is not empty,
  // separate it by a space from the trailing text
  public static String separateTrailing(Object object)
  {
    String text = object.toString();
    return text.equals("") ? "" : text+" ";
  }
#}

// ------------------------------------------------------------------------
// printer
// ------------------------------------------------------------------------
printer
{
  domain Session
  {
    case SessionStats[stats] →
      # _result = stats.toString(); #;
  }
  domain Stats
  {
    case SingleStat[stat] → # _result = stat.toString(); #;
    case MultipleStats[stat,stats] → # _result = stat + "\n" + stats; #;
  }
  domain Stat
  {
    case StatCreateTable[stat] → # _result = stat.toString(); #;
    case StatDropTable[stat] → # _result = stat.toString(); #;
    case StatInsert[stat] → # _result = stat.toString(); #;
    case StatUpdate[stat] → # _result = stat.toString(); #;
    case StatDelete[stat] → # _result = stat.toString(); #;
    case StatSelect[stat] → # _result = stat.toString(); #;
  }
  domain CreateTableStat
  {
    case CreateTable[id,cdecls,annot] →
      # _result = "CREATE TABLE " + id + "(" + cdecls + ")" + annot + ";"; #;
  }
  domain ColumnDecls
  {
```

```
  case SingleColumnDecl[cdecl] →
    # _result = cdecl.toString(); #;
  case MultipleColumnDecls[cdecl,cdecls] →
    # _result = cdecl + "," + cdecls; #;
}
domain ColumnDecl
{
  case TheColumnDecl[id,ctexp,consts] →
    # _result = id + " " + ctexp + separate(consts); #;
}
domain ColumnTypeExp
{
  case ColumnTypeInt[num] → # _result = "INT(" + num + ")"; #;
  case ColumnTypeVarChar[num] → # _result = "VARCHAR(" + num + ")"; #;
}
domain Constraints
{
  case NoConstraint →
    # _result = ""; #;
  case SomeConstraint[c,cs] →
    # _result = c.toString() + separate(cs); #;
}
domain Constraint
{
  case Default[lit] → # _result = "DEFAULT " + lit; #;
  case NotNull → # _result = "NOT NULL"; #;
  case Unique → # _result = "UNIQUE"; #;
  case PrimaryKey → # _result = "PRIMARY KEY"; #;
}
domain DropTableStat
{
  case DropTable[id] → # _result = "DROP TABLE " + id + ";"; #;
}
domain InsertStat
{
  case Insert[id,iclause,texp,annot] →
    # _result = "INSERT INTO " + id + separate(iclause)
        + " " + texp + annot + ";"; #;
}
domain InsertColumnClause
{
  case NoInsertColumnClause → # _result = ""; #;
  case SomeInsertColumnClause[ids] → # _result = ids.toString(); #;
}
domain IDs
{
  case SingleID[id] → # _result = id.toString(); #;
  case MultipleIDs[id,ids] -> # _result = id + "," + ids; #;
}
domain UpdateStat
{
  case Update[id,ass,where,annot] → # _result = "UPDATE " + id + " SET " + ass +
    separate(where) + annot + ";"; #;
}
```

```
domain Assignments
{
  case SingleAssignment[a] → # _result = a.toString(); #;
  case MultipleAssignments[a,as] → # _result = a + ", " + as; #;
}
domain Assignment
{
  case TheAssignment[id,exp,annot] → # _result = id + " = " + exp + annot; #;
}
domain DeleteStat
{
  case Delete[id,where,annot] → # _result = "DELETE FROM " + id +
    separate(where) + annot + ";"; #;
}
domain SelectStat
{
  case Select[texp,annot] → # _result = texp + "" + annot + ";"; #;
}
domain TableExp
{
  case TableExpValuesExp[vexp] → # _result = vexp.toString(); #;
  case TableExpSelectExp[sexp] → # _result = sexp.toString(); #;
  case TableExpCombination[texp1,op,dist,texp2] →
    # _result = "(" + texp1 + ") " + op +
      separate(dist) + " (" + texp2 + ")"; #;
}
domain SetOp
{
  case SetUnion → # _result = "UNION"; #;
  case SetIntersect → # _result = "INTERSECT"; #;
  case SetExcept → # _result = "EXCEPT"; #;
}
domain DistinctClause
{
  case DistinctNone → # _result = ""; #;
  case DistinctAll → # _result = "ALL"; #;
  case DistinctDistinct → # _result = "DISTINCT"; #;
}
domain ValuesExp
{
  case TheValuesExp[rexps] → # _result = "VALUES " + rexps; #;
}
domain RowExps
{
  case SingleRowExp[rexp] → # _result = rexp.toString(); #;
  case MultipleRowExps[rexp,rexps] → # _result = rexp + "," + rexps; #;
}
domain RowExp
{
  case TheRowExp[exps] -> # _result = "(" + exps + ")"; #;
}
domain SelectExp
{
  case TheSelectExp[distinct,select,from,where,group,having,order] →
```

```
    # _result = "SELECT" + separate(distinct) + " " +
       select + separate(from) + separate(where) +
       separate(group) + separate(having) +
       separate(order); #;
}
domain FromClause
{
  case NoFromClause → # _result = ""; #;
  case SomeFromClause[table] → # _result = "FROM " + table.toString(); #;
}
domain TableD
{
  case TableId[id,as,annot] → # _result = id + separate(as) + annot; #;
  case TableTableExp[exp,as] → # _result = exp + separate(as); #;
  case TableJoin[table1,join,table2,cond,annot] →
    # _result = table1 + " " + join + " " + table2 + " " + cond + annot; #;
  case TableCrossJoin[table1,cjoin,table2] →
    # _result = table1 + cjoin.toString() + table2; #;
}
domain AsClause
{
  case NoAsClause → # _result = ""; #;
  case SomeAsClause[id] → # _result = id; #;
  case SomeAsClauseAs[id] → # _result = "AS " + id; #;
}
domain JoinClause
{
  case Join → # _result = "JOIN"; #;
  case InnerJoin → # _result = "INNER JOIN"; #;
  case LeftJoin → # _result = "LEFT JOIN"; #;
  case LeftOuterJoin → # _result = "LEFT OUTER JOIN"; #;
  case RightJoin → # _result = "RIGHT JOIN"; #;
  case RightOuterJoin → # _result = "RIGHT OUTER JOIN"; #;
  case FullJoin → # _result = "FULL JOIN"; #;
  case FullOuterJoin → # _result = "FULL OUTER JOIN"; #;
}
domain JoinCondition
{
  case JoinOn[exp] → # _result = "ON " + exp; #;
  case JoinUsing[ids,annot] → # _result = "USING (" + ids + ")" + annot; #;
}
domain CrossJoinClause
{
  case CrossJoin → # _result = " CROSS JOIN "; #;
  case CommaJoin → # _result = ","; #;
}
domain WhereClause
{
  case NoWhere → # _result = ""; #;
  case SomeWhere[exp] → # _result = "WHERE " + exp; #;
}
domain GroupByClause
{
  case NoGroupBy → # _result = ""; #;
```

```
  case SomeGroupBy[crefs,annot] → # _result = "GROUP BY " + crefs + annot; #;
}
domain HavingClause
{
  case NoHaving → # _result = ""; #;
  case SomeHaving[exp] → # _result = "HAVING " + exp; #;
}
domain SelectClause
{
  case TheSelectClause[cexps,annot] → # _result = cexps.toString() + annot; #;
}
domain ColumnExps
{
  case SingleColumnExp[cexp] → # _result = cexp.toString(); #;
  case MultipleColumnExps[cexp,cexps] → # _result = cexp + "," + cexps; #;
}
domain ColumnExp
{
  case ColumnExpAll → # _result = "*"; #;
  case ColumnExpId[id,annot] → # _result = id + ".*" + annot; #;
  case ColumnExpExp[exp,as] → # _result = exp + separate(as); #;
}
domain OrderByClause
{
  case OrderByNone -> # _result = ""; #;
  case OrderBySome[ocols,annot] → # _result = "ORDER BY " + ocols + annot; #;
}
domain OrderByColumns
{
  case SingleOrderByColumn[ocol,annot] →
    # _result = ocol.toString() + annot; #;
  case MultipleOrderByColumns[ocol,ocols,annot] →
    # _result = ocol + "," + ocols + annot; #;
}
domain OrderByColumn
{
  case TheOrderByColumn[cref,order,annot] →
    # _result = cref + separate(order) + annot; #;
}
domain OrderClause
{
  case OrderNone → # _result = ""; #;
  case OrderAsc → # _result = "ASC"; #;
  case OrderDesc → # _result = "DESC"; #;
}
domain Exps
{
  case SingleExp[exp] → # _result = exp.toString(); #;
  case MultipleExps[exp,exps] → # _result = exp + "," + exps; #;
}
domain Exp
{
  case ExpMetaVar[id] →
    # _result = "{" + id + "}"; #;
```

```
  case ExpLiteral[lit,annot] →
    # _result = lit.toString() + annot; #;
  case ExpRef[cref,annot] →
    # _result = cref.toString() + annot; #;
  case ExpUnary[op,exp] →
    # _result = "(" + op.toString() + exp + ")"; #;
  case ExpBinary[exp1,op,exp2] →
    # _result = "(" + exp1 + op.toString() + exp2 + ")"; #;
  case ExpExps[exps] →
    # _result = "(" + exps + ")"; #;
  case ExpSelect[sexp] →
    # _result = "(" + sexp + ")"; #;
  case ExpBetween[exp,not,exp1,exp2] →
    # _result = "(" + exp + separate(not) +
      " BETWEEN " + exp1 + " AND " + exp2 + ")"; #;
  case ExpIsNull[exp,not] →
    # _result = "(" + exp + " IS" + separate(not) + " NULL)"; #;
  case ExpIsTrue[exp,not] →
    # _result = "(" + exp + " IS" + separate(not) + " TRUE)"; #;
  case ExpIsFalse[exp,not] →
    # _result = "(" + exp + " IS" + separate(not) + " FALSE)"; #;
  case ExpIsDistinct[exp1,not,exp2] →
    # _result = "(" + exp1 + " IS" + separate(not) +
      " DISTINCT FROM " + exp2 + ")"; #;
  case ExpLike[exp,str,annot] →
    # _result = "(" + exp + " LIKE " + str + annot + ")"; #;
  case ExpCase[when,els] →
    # _result = "CASE " + when + separate(els) + " END"; #;
  case ExpCaseExp[exp,when,els] →
    # _result = "CASE " + exp + " " + when + separate(els) + " END"; #;
  case ExpNullIf[exp1,exp2] →
    # _result = "NULLIF(" + exp1 + "," + exp2 + ")"; #;
  case ExpCoalesce[exps] →
    # _result = "COALESCE(" + exps + ")"; #;
  case ExpExists[sexp] →
    # _result = "EXISTS(" + sexp + ")"; #;
  case ExpIn[exp,not,exps] →
    # _result = exp + separate(not) + " IN (" + exps + ")"; #;
  case ExpInSelect[exp,not,sexp] →
    # _result = exp + separate(not) + " IN (" + sexp + ")"; #;
  case ExpQuantified[exp,op,quant,sexp] →
    # _result = exp + " " + op + " " + quant + " (" + sexp + ")"; #;
  case ExpAggregateExp[aexp] →
    # _result = aexp.toString(); #;
}
domain ColumnRef
{
  case PlainRef[id] → # _result = id; #;
  case TableRef[id1,id2] → # _result = id1 + "." + id2; #;
}
domain ColumnRefs
{
  case SingleColumnRef[cref] → # _result = cref.toString(); #;
  case MultipleColumnRefs[cref,crefs] → # _result = cref + "," + crefs; #;
```

```
}
domain Literal
{
  case LiteralNUM[num] → # _result = num; #;
  case LiteralSTR[str] → # _result = str; #;
  case LiteralNull → # _result = "NULL"; #;
  case LiteralTrue → # _result = "TRUE"; #;
  case LiteralFalse → # _result = "FALSE"; #;
}
domain UnaryOp
{
  case UnaryNot → # _result = "NOT "; #;
  case UnaryMinus → # _result = "-"; #;
}
domain BinaryOp
{
  case BinaryPlus → # _result = "+"; #;
  case BinaryMinus → # _result = "-"; #;
  case BinaryTimes → # _result = "*"; #;
  case BinaryDiv → # _result = "/"; #;
  case BinaryMod → # _result = "%"; #;
  case BinaryBitAnd → # _result = "&"; #;
  case BinaryBitOr → # _result = "|"; #;
  case BinaryBitExclOr → # _result = "^"; #;
  case BinaryEqual → # _result = " = "; #;
  case BinaryGreater → # _result = " > "; #;
  case BinaryGreaterEqual → # _result = " >= "; #;
  case BinaryLess → # _result = " < "; #;
  case BinaryLessEqual → # _result = " <= "; #;
  case BinaryNotEqual → # _result = " <> "; #;
  case BinaryAnd → # _result = " AND "; #;
  case BinaryOr → # _result = " OR "; #;
}
domain NotClause
{
  case NoNot → # _result = ""; #;
  case SomeNot → # _result = "NOT"; #;
}
domain Quantifier
{
  case QuantifierAny → # _result = "ANY"; #;
  case QuantifierSome → # _result = "SOME"; #;
  case QuantifierAll → # _result = "ALL"; #;
}
domain WhenClauses
{
  case SingleWhen[when] → # _result = when.toString(); #;
  case MultipleWhens[when,whens] → # _result = when + ", " + whens; #;
}
domain WhenClause
{
  case TheWhenClause[exp1,exp2] →
    # _result = "WHEN " + exp1 + " THEN " + exp2; #;
}
```

```
  domain ElseClause
  {
    case NoElse → # _result = ""; #;
    case SomeElse[exp] → # _result = "ELSE " + exp; #;
  }
  domain AggregateExp
  {
    case AggregateExpFun[fun,arg] → # _result = fun + "(" + arg + ")"; #;
    case AggregateExpCount → # _result = "COUNT(*)"; #;
  }
  domain AggregateArg
  {
    case TheAggregateArg[distinct,exp] →
      # _result = separateTrailing(distinct) + exp; #;
  }
  domain AggregateFun
  {
    case AggregateMin → # _result = "MIN"; #;
    case AggregateMax → # _result = "MAX"; #;
    case AggregateSum → # _result = "SUM"; #;
    case AggregateAvg → # _result = "AVG"; #;
    case AggregateCount → # _result = "COUNT"; #;
  }
}

// ---------------------------------------------------------------------
// parser
// ---------------------------------------------------------------------
parser antlr4
{
  domain Session
  {
    case # stats=dStats EOF # →
      SessionStats[stats];
  }
  domain Stats
  {
    case # stat=dStat # → SingleStat[stat];
    case # stat=dStat stats=dStats # → MultipleStats[stat,stats];
  }
  domain Stat
  {
    case # cstat=dCreateTableStat # → StatCreateTable[cstat];
    case # dstat=dDropTableStat # → StatDropTable[dstat];
    case # istat=dInsertStat # → StatInsert[istat];
    case # ustat=dUpdateStat # → StatUpdate[ustat];
    case # estat=dDeleteStat # → StatDelete[estat];
    case # sstat=dSelectStat # → StatSelect[sstat];
  }
  domain CreateTableStat
  {
    case # 'CREATE' 'TABLE' id=dID '(' cdecls=dColumnDecls ')' ';' # →
      // CreateTable[id,cdecls];
      # $_result = new CreateTableStat.CreateTable($id._result,
```

```
        $cdecls._result, new Annotation()); #;
}
domain ColumnDecls
{
  case # cdecl=dColumnDecl # → SingleColumnDecl[cdecl];
  case # cdecl=dColumnDecl ',' cdecls=dColumnDecls # ->
    MultipleColumnDecls[cdecl,cdecls];
}
domain ColumnDecl
{
  case # id=dID ctexp=dColumnTypeExp constraints=dConstraints # →
    TheColumnDecl[id,ctexp,constraints];
}
domain ColumnTypeExp
{
  case # 'INT' '(' num=dNUM ')' # → ColumnTypeInt[num];
  case # 'VARCHAR' '(' num=dNUM ')' # → ColumnTypeVarChar[num];
}
domain Constraints
{
  case # /* empty */ # → NoConstraint;
  case # constraint=dConstraint constraints=dConstraints # →
    SomeConstraint[constraint,constraints];
}
domain Constraint
{
  case # 'DEFAULT' lit=dLiteral # → Default[lit];
  case # 'NOT' 'NULL' # → NotNull;
  case # 'UNIQUE' # → Unique;
  case # 'PRIMARY KEY' # → PrimaryKey;
}
domain DropTableStat
{
  case # 'DROP' 'TABLE' id=dID ';' # → DropTable[id];
}
domain InsertStat
{
  case # 'INSERT' 'INTO' id=dID icol=dInsertColumnClause
         texp=dTableExp ';' # → // Insert[id,icol,texp];
    # $_result = new InsertStat.Insert($id._result,
        $icol._result, $texp._result, new Annotation()); #;
}
domain InsertColumnClause
{
  case # /* empty */ # → NoInsertColumnClause;
  case # '(' ids=dIDs ')' # → SomeInsertColumnClause[ids];
}
domain IDs
{
  case # id=dID # → SingleID[id];
  case # id=dID ',' ids=dIDs # → MultipleIDs[id,ids];
}
domain UpdateStat
{
```

```
  case # 'UPDATE' id=dID 'SET' as=dAssignments where=dWhereClause ';' # →
    // Update[id,as,where];
    # $_result = new UpdateStat.Update($id._result,
        $as._result, $where._result, new Annotation()); #;
}
domain Assignments
{
  case # a=dAssignment # → SingleAssignment[a];
  case # a=dAssignment ',' as=dAssignments # → MultipleAssignments[a,as];
}
domain Assignment
{
  case # id=dID '=' exp=dExp # → // TheAssignment[id,exp];
    # $_result = new Assignment.TheAssignment($id._result,
        $exp._result, new Annotation()); #;
}
domain DeleteStat
{
  case # 'DELETE' 'FROM' id=dID where=dWhereClause ';' # →
    // Delete[id,where];
    # $_result = new DeleteStat.Delete($id._result,
        $where._result, new Annotation()); #;
}
domain SelectStat
{
  case # texp=dTableExp ';' # → // Select[texp];
    # $_result = new SelectStat.Select($texp._result,
        new Annotation()); #;
}
domain TableExp
{
  case # vexp=dValuesExp # → TableExpValuesExp[vexp];
  case # sexp=dSelectExp # → TableExpSelectExp[sexp];
  case # '(' texp1=dTableExp ')' op=dSetOp dist=dDistinctClause
    '(' texp2=dTableExp ')' # →
    TableExpCombination[texp1,op,dist,texp2];
}
domain SetOp
{
  case # 'UNION' # → SetUnion;
  case # 'INTERSECT' # → SetIntersect;
  case # 'EXCEPT' # → SetExcept;
}
domain DistinctClause
{
  case # /* empty */ # → DistinctNone;
  case # 'ALL' # -> DistinctAll;
  case # 'DISTINCT' # → DistinctDistinct;
}
domain ValuesExp
{
  case # 'VALUES' rexps=dRowExps # → TheValuesExp[rexps];
}
domain RowExps
```

```
{
  case # rexp=dRowExp # → SingleRowExp[rexp];
  case # rexp=dRowExp ',' rexps=dRowExps # → MultipleRowExps[rexp,rexps];
}
domain RowExp
{
  case # '(' exps=dExps ')' # → TheRowExp[exps];
}
domain SelectExp
{
  case # 'SELECT' distinct=dDistinctClause select=dSelectClause
    from=dFromClause where=dWhereClause
    group=dGroupByClause having=dHavingClause order=dOrderByClause # →
    TheSelectExp[distinct,select,from,where,group,having,order];
}
domain FromClause
{
  case # /* empty */ # → NoFromClause;
  case # 'FROM' table=dTableD # → SomeFromClause[table];
}
domain TableD
{
  case # id=dID as=dAsClause # → // TableId[id,as];
   # $_result = new TableD.TableId($id._result,
        $as._result, new Annotation()); #;
  case # '(' texp=dTableExp ')' as=dAsClause # → TableTableExp[texp,as];
  case # t1=dTableD join=dJoinClause t2=dTableD cond=dJoinCondition # →
    // TableJoin[t1,join,t2,cond];
   # $_result = new TableD.TableJoin($t1._result,
        $join._result, $t2._result, $cond._result, new Annotation()); #;
  case # t1=dTableD cjoin=dCrossJoinClause t2=dTableD # →
    TableCrossJoin[t1,cjoin,t2];
}
domain AsClause
{
  case # /* empty */ # → NoAsClause;
  case # id=dID # → SomeAsClause[id];
  case # 'AS' id=dID # → SomeAsClauseAs[id];
}
domain JoinClause
{
  case # 'JOIN' # -> Join;
  case # 'INNER' 'JOIN' # -> InnerJoin;
  case # 'LEFT' 'JOIN' # -> LeftJoin;
  case # 'LEFT' 'OUTER' 'JOIN' # -> LeftOuterJoin;
  case # 'RIGHT' 'JOIN' # -> RightJoin;
  case # 'RIGHT' 'OUTER' 'JOIN' # -> RightOuterJoin;
  case # 'FULL' 'JOIN' # -> FullJoin;
  case # 'FULL' 'OUTER' 'JOIN' # -> FullOuterJoin;
}
domain JoinCondition
{
  case # 'ON' exp=dExp # → JoinOn[exp];
  case # 'USING' '(' ids=dIDs ')' # → // JoinUsing[ids];
```

```
     # $_result = new JoinCondition.JoinUsing($ids._result,
          new Annotation()); #;
}
domain CrossJoinClause
{
  case # 'CROSS' 'JOIN' # → CrossJoin;
  case # ',' # → CommaJoin;
}
domain WhereClause
{
  case # /* empty */ # → NoWhere;
  case # 'WHERE' exp=dExp # -> SomeWhere[exp];
}
domain GroupByClause
{
  case # /* empty */ # → NoGroupBy;
  case # 'GROUP' 'BY' crefs=dColumnRefs # -> // SomeGroupBy[crefs];
    # $_result = new GroupByClause.SomeGroupBy($crefs._result,
        new Annotation()); #;
}
domain HavingClause
{
  case # /* empty */ # → NoHaving;
  case # 'HAVING' exp=dExp # -> SomeHaving[exp];
}
domain SelectClause
{
  case # cexps=dColumnExps # → // TheSelectClause[cexps];
    # $_result = new SelectClause.TheSelectClause($cexps._result,
        new Annotation()); #;
}
domain ColumnExps
{
  case # cexp=dColumnExp # → SingleColumnExp[cexp];
  case # cexp=dColumnExp ',' cexps=dColumnExps # →
    MultipleColumnExps[cexp,cexps];
}
domain ColumnExp
{
  case # '*' # → ColumnExpAll;
  case # id=dID '.' '*' # → // ColumnExpId[id];
    # $_result = new ColumnExp.ColumnExpId($id._result,
        new Annotation()); #;
  case # exp=dExp as=dAsClause # → ColumnExpExp[exp,as];
}
domain OrderByClause
{
  case # /* empty */ # → OrderByNone;
  case # 'ORDER' 'BY' ocols=dOrderByColumns # -> // OrderBySome[ocols];
    # $_result = new OrderByClause.OrderBySome($ocols._result,
        new Annotation()); #;
}
domain OrderByColumns
{
```

```
  case # ocol=dOrderByColumn # → // SingleOrderByColumn[ocol];
    # $_result = new OrderByColumns.SingleOrderByColumn($ocol._result,
        new Annotation()); #;
  case # ocol=dOrderByColumn ',' ocols=dOrderByColumns # →
    // MultipleOrderByColumns[ocol,ocols];
    # $_result = new OrderByColumns.MultipleOrderByColumns($ocol._result,
        $ocols._result, new Annotation()); #;
}
domain OrderByColumn
{
  case # cref=dColumnRef order=dOrderClause # → // TheOrderByColumn[cref,order];
    # $_result = new OrderByColumn.TheOrderByColumn($cref._result,
        $order._result, new Annotation()); #;
}
domain OrderClause
{
  case # /* empty */ # → OrderNone;
  case # 'ASC' # -> OrderAsc;
  case # 'DESC' # → OrderDesc;
}
domain Exps
{
  case # exp=dExp # → SingleExp[exp];
  case # exp=dExp ',' exps=dExps # → MultipleExps[exp,exps];
}
domain Exp
{
  case # '{' id=dID '}' # → ExpMetaVar[id];
  case # lit=dLiteral # → // ExpLiteral[lit];
    # $_result = new Exp.ExpLiteral($lit._result, new Annotation()); #;
  case # cref=dColumnRef # → // ExpRef[cref];
    # $_result = new Exp.ExpRef($cref._result, new Annotation()); #;
  // case # uop=dUnaryOp exp=dExp # → ExpUnary[uop,exp];
  // case # exp1=dExp bop=dBinaryOp exp2=dExp # → ExpBinary[exp1,bop,exp2];
  // place rules for higher-precedence operators first
  case # exp1=dExp op='&' exp2=dExp # →
    # BinaryOp bop = new BinaryOp.BinaryBitAnd();
      $_result = new Exp.ExpBinary($exp1._result,bop,$exp2._result); #;
  case # exp1=dExp ( op='|' | op='^' ) exp2=dExp # →
    # BinaryOp bop = SubSQL_parser.parseBinaryOp($op.getText());
      $_result = new Exp.ExpBinary($exp1._result,bop,$exp2._result); #;
  case # op='-' exp=dExp # →
    # UnaryOp uop = new UnaryOp.UnaryMinus();
      $_result = new Exp.ExpUnary(uop,$exp._result); #;
  case # exp1=dExp ( op='*' | op='/' | op='%' ) exp2=dExp # →
    # BinaryOp bop = SubSQL_parser.parseBinaryOp($op.getText());
      $_result = new Exp.ExpBinary($exp1._result,bop,$exp2._result); #;
  case # exp1=dExp ( op='+' | op='-' ) exp2=dExp # →
    # BinaryOp bop = SubSQL_parser.parseBinaryOp($op.getText());
      $_result = new Exp.ExpBinary($exp1._result,bop,$exp2._result); #;
  case # exp1=dExp ( op='=' | op='>' | op='>=' | op='<' | op='<=' |
                    op='<>' ) exp2=dExp # →
    # BinaryOp bop = SubSQL_parser.parseBinaryOp($op.getText());
      $_result = new Exp.ExpBinary($exp1._result,bop,$exp2._result); #;
```

```
  case # op='NOT' exp=dExp # →
    # UnaryOp uop = new UnaryOp.UnaryNot();
      $_result = new Exp.ExpUnary(uop,$exp._result); #;
  case # exp1=dExp op='AND' exp2=dExp # →
    # BinaryOp bop = new BinaryOp.BinaryAnd();
      $_result = new Exp.ExpBinary($exp1._result,bop,$exp2._result); #;
  case # exp1=dExp op='OR' exp2=dExp # →
    # BinaryOp bop = new BinaryOp.BinaryOr();
      $_result = new Exp.ExpBinary($exp1._result,bop,$exp2._result); #;
  case # '(' exps=dExps ')' # → ExpExps[exps];
  case # '(' sexp=dSelectExp ')' # → ExpSelect[sexp];
  case # exp=dExp not=dNotClause 'BETWEEN' exp1=dExp 'AND' exp2=dExp # →
    ExpBetween[exp,not,exp1,exp2];
  case # exp=dExp 'IS' not=dNotClause 'NULL' # → ExpIsNull[exp,not];
  case # exp=dExp 'IS' not=dNotClause 'TRUE' # → ExpIsTrue[exp,not];
  case # exp=dExp 'IS' not=dNotClause 'FALSE' # → ExpIsFalse[exp,not];
  case # exp1=dExp 'IS' not=dNotClause 'DISTINCT' 'FROM' exp2=dExp # →
    ExpIsDistinct[exp1,not,exp2];
  case # exp=dExp 'LIKE' str=dSTR # → // ExpLike[exp,str];
    # $_result = new Exp.ExpLike($exp._result,
        $str._result,new Annotation()); #;
  case # 'CASE' when=dWhenClauses els=dElseClause 'END' # →
    ExpCase[when,els];
  case # 'CASE' exp=dExp when=dWhenClauses els=dElseClause 'END' # →
    ExpCaseExp[exp,when,els];
  case # 'NULLIF' '(' exp1=dExp ',' exp2=dExp ')' # → ExpNullIf[exp1,exp2];
  case # 'COALESCE' '(' exps=dExps ')' # → ExpCoalesce[exps];
  case # 'EXISTS' '(' sexp=dSelectExp ')' # → ExpExists[sexp];
  case # exp1=dExp not=dNotClause 'IN' '(' exps=dExps ')' # →
    ExpIn[exp1,not,exps];
  case # exp=dExp not=dNotClause 'IN' '(' sexp=dSelectExp ')' # →
    ExpInSelect[exp,not,sexp];
  case # exp=dExp bop=dBinaryOp quant=dQuantifier '(' sexp=dSelectExp ')' # →
    ExpQuantified[exp,bop,quant,sexp];
  case # aexp=dAggregateExp # →
    ExpAggregateExp[aexp];
}
domain ColumnRef
{
  case # id=dID # → PlainRef[id];
  case # id1=dID '.' id2=dID # → TableRef[id1,id2];
}
domain ColumnRefs
{
  case # cref=dColumnRef # → SingleColumnRef[cref];
  case # cref=dColumnRef ',' crefs=dColumnRefs # →
    MultipleColumnRefs[cref,crefs];
}
domain Literal
{
  case # num=dNUM # → LiteralNUM[num];
  case # str=dSTR # → LiteralSTR[str];
  case # 'NULL' # → LiteralNull;
  case # 'TRUE' # → LiteralTrue;
```

```
  case # 'FALSE' # → LiteralFalse;
}
domain UnaryOp
{
  case # 'NOT' # → UnaryNot;
  case # '-' # → UnaryMinus;
}
domain BinaryOp
{
  case # '+' # → BinaryPlus;
  case # '-' # → BinaryMinus;
  case # '*' # → BinaryTimes;
  case # '/' # → BinaryDiv;
  case # '%' # → BinaryMod;
  case # '&' # → BinaryBitAnd;
  case # '|' # → BinaryBitOr;
  case # '^' # → BinaryBitExclOr;
  case # '=' # → BinaryEqual;
  case # '>' # → BinaryGreater;
  case # '>=' # → BinaryGreaterEqual;
  case # '<' # → BinaryLess;
  case # '<=' # → BinaryLessEqual;
  case # '<>' # → BinaryNotEqual;
  case # 'AND' # → BinaryAnd;
  case # 'OR' # → BinaryOr;
}
domain NotClause
{
  case # /* empty */ # → NoNot;
  case # 'NOT' # -> SomeNot;
}
domain Quantifier
{
  case # 'ANY' # -> QuantifierAny;
  case # 'SOME' # -> QuantifierSome;
  case # 'ALL' # -> QuantifierAll;
}
domain WhenClauses
{
  case # when=dWhenClause # → SingleWhen[when];
  case # when=dWhenClause ',' whens=dWhenClauses # →
    MultipleWhens[when,whens];
}
domain WhenClause
{
  case # 'WHEN' exp1=dExp 'THEN' exp2=dExp # → TheWhenClause[exp1,exp2];
}
domain ElseClause
{
  case # /* empty */ # → NoElse;
  case # 'ELSE' exp=dExp # → SomeElse[exp];
}
domain AggregateExp
{
```

```
    case # fun=dAggregateFun '(' arg=dAggregateArg ')' # → AggregateExpFun[fun,arg];
    case # 'COUNT' '(' '*' ')' # → AggregateExpCount;
  }
  domain AggregateArg
  {
    case # dist=dDistinctClause exp=dExp # → TheAggregateArg[dist,exp];
  }
  domain AggregateFun
  {
    case # 'MIN' # → AggregateMin;
    case # 'MAX' # → AggregateMax;
    case # 'SUM' # → AggregateSum;
    case # 'AVG' # → AggregateAvg;
    case # 'COUNT' # → AggregateCount;
  }
}

// ------------------------------------------------------------------------
// type system
// ------------------------------------------------------------------------
judgment #Env# ⊢ Session: session(#Env#)
{
  inference env0 ⊢ SessionStats[Stats]: session(env)
  {
    env0 ⊢ Stats: stats(env);
  }
}
judgement #Env# ⊢ Stats: stats(#Env#)
{
  inference env0 ⊢ SingleStat[Stat]: stats(env)
  {
    env0 ⊢ Stat: stat(env1);
    env = #Env.setDBEnv(env1)#;
  }
  inference env0 ⊢ MultipleStats[Stat,Stats]: stats(env)
  {
    env0 ⊢ Stat: stat(env1);
    env2: #Env# = #Env.setDBEnv(env1)#;
    env2 ⊢ Stats: stats(env);
  }
}
judgement #Env# ⊢ Stat: stat(#Env#)
{
  inference env0 ⊢ StatCreateTable[CreateTableStat]: stat(env)
  {
    env0 ⊢ CreateTableStat: createTableStat(env);
  }
  inference env0 ⊢ StatDropTable[DropTableStat]: stat(env)
  {
    env0 ⊢ DropTableStat: dropTableStat(env);
  }
  inference env0 ⊢ StatInsert[InsertStat]: stat(env)
  {
    env0 ⊢ InsertStat: insertStat; env = env0;
```

```
  }
  inference env0 ⊢ StatUpdate[UpdateStat]: stat(env)
  {
    env0 ⊢ UpdateStat: updateStat; env = env0;
  }
  inference env0 ⊢ StatDelete[DeleteStat]: stat(env)
  {
    env0 ⊢ DeleteStat: deleteStat; env = env0;
  }
  inference env0 ⊢ StatSelect[SelectStat]: stat(env)
  {
    env1: #Env# = #Env.clearTableEnv(env0)#;
    env1 ⊢ SelectStat: selectStat; env = env0;
  }
}
judgement #Env# ⊢ CreateTableStat: createTableStat(#Env#)
{
  inference env0 ⊢ CreateTable[ID,ColumnDecls,Annotation]: createTableStat(env)
  {
    id: #Id# = #new Id(ID)#;
    # check(env0.tenv().map().get(id) == null,
        "duplicate declaration of table " + ID); #
    csyms0: #ColumnSymbols# = #new ColumnSymbols()#;
    id,csyms0 ⊢ ColumnDecls: columnDecls(csyms);
    # TableSymbol tsym = new TableSymbol(id,csyms); #
    env = #env0.put(id,tsym)#;
    # Annotation.setColumnSymbols(csyms); #
  }
}
judgement #Id#,#ColumnSymbols# ⊢ ColumnDecls: columnDecls(#ColumnSymbols#)
{
  inference id,csyms0 ⊢ SingleColumnDecl[ColumnDecl]: columnDecls(csyms)
  {
    id,csyms0 ⊢ ColumnDecl: columnDecl(csyms);
  }
  inference id,csyms0 ⊢ MultipleColumnDecls[ColumnDecl,ColumnDecls]:
    columnDecls(csyms)
  {
    id,csyms0 ⊢ ColumnDecl: columnDecl(csyms1);
    id,csyms1 ⊢ ColumnDecls: columnDecls(csyms);
  }
}
judgement #Id#,#ColumnSymbols# ⊢ ColumnDecl: columnDecl(#ColumnSymbols#)
{
  inference tid,csyms0 ⊢ TheColumnDecl[ID,ColumnTypeExp,Constraints]:
    columnDecl(csyms)
  {
    cid: #Id# = #new Id(ID)#;
    # check(!exists(csyms0.list(), (ColumnSymbol csym)->
               csym.id().name().equals(cid.name())),
            "duplicate declaration of column " + cid.name()); #
    ⊢ ColumnTypeExp: columnTypeExp(ctype);
    cset0: #ConstraintSet# = #new ConstraintSet()#;
    cset0,ctype ⊢ Constraints: constraints(cset);
```

```
    csym: #ColumnSymbol# = #new ColumnSymbol(cid,ctype,cset,
      new IdOption(Optional.of(tid)))#;
    csyms = #csyms0.add(csym)#;
  }
}
judgement ⊢ ColumnTypeExp: columnTypeExp(#ColumnType#)
{
   inference ⊢ ColumnTypeInt[NUM]: columnTypeExp(ctype)
   {
     n: #int# = #Integer.parseInt(NUM)#;
     ctype = #new ColumnType.Int(n)#;
   }
   inference ⊢ ColumnTypeVarChar[NUM]: columnTypeExp(ctype)
   {
     n: #int# = #Integer.parseInt(NUM)#;
     ctype = #new ColumnType.VarChar(n)#;
   }
}
judgement #ConstraintSet#,#ColumnType# ⊢ Constraints: constraints(#ConstraintSet#)
{
  inference cset0,ctype ⊢ NoConstraint: constraints(cset)
  {
    cset = cset0;
  }
  inference cset0,ctype ⊢ SomeConstraint[Constraint,Constraints]: constraints(cset)
  {
    cset0,ctype ⊢ Constraint: constraint(cset1);
    cset1,ctype ⊢ Constraints: constraints(cset);
  }
}
judgement #ConstraintSet#,#ColumnType# ⊢ Constraint: constraint(#ConstraintSet#)
{
  inference cset0,ctype ⊢ Default[Literal]: constraint(cset)
  {
    # check(!exists(cset0.set(),(ConstraintSymbol cons)->
              cons instanceof ConstraintSymbol.Default),
            "duplicate declaration of constraint DEFAULT"); #
    ⊢ Literal: literal(type,value);
    # check(ctype instanceof ColumnType.Int ? type instanceof Type.Int :
            ctype instanceof ColumnType.VarChar ? type instanceof Type.String :
            false, "default value " + Literal + " has type " + type +
            " which is incompatible with column type " + ctype.type()); #
    cset = #cset0.add(new ConstraintSymbol.Default(value))#;
  }
  inference cset0,ctype ⊢ NotNull: constraint(cset)
  {
    # check(!exists(cset0.set(),(ConstraintSymbol cons)->
              cons instanceof ConstraintSymbol.Notnull),
        "duplicate declaration of constraint NOT NULL " +
        "(may also clash with constraint PRIMARY KEY)"); #
    cset = #cset0.add(new ConstraintSymbol.Notnull())#;
  }
  inference cset0,ctype ⊢ Unique: constraint(cset)
  {
```

```
    # check(!exists(cset0.set(),(ConstraintSymbol cons)->
              cons instanceof ConstraintSymbol.Unique),
        "duplicate declaration of constraint UNIQUE " +
        "(may also clash with constraint PRIMARY KEY)"); #
    cset = #cset0.add(new ConstraintSymbol.Unique())#;
  }
  inference cset0,ctype ⊢ PrimaryKey: constraint(cset)
  {
    # check(!exists(cset0.set(),(ConstraintSymbol cons)->
              (cons instanceof ConstraintSymbol.Notnull) ||
              (cons instanceof ConstraintSymbol.Unique)),
        "duplicate declaration of constraint PRIMARY KEY " +
        "(may also clash with constraint NOT NULL or UNIQUE"); #
    cset = #cset0.add(new ConstraintSymbol.Notnull(),
                      new ConstraintSymbol.Unique())#;
  }
}
judgement #Env# ⊢ DropTableStat: dropTableStat(#Env#)
{
  inference env0 ⊢ DropTable[ID]: dropTableStat(env)
  {
    id: #Id# = #new Id(ID)#;
    # check(env0.tenv().map().containsKey(id), "undeclared table " + id.name()); #
    env = #env0.removeTable(id)#;
  }
}
judgement #Env# ⊢ InsertStat: insertStat
{
  inference env0 ⊢ Insert[ID,InsertColumnClause,TableExp,Annotation]: insertStat
  {
    id: #Id# = #new Id(ID)#;
    tsym: #TableSymbol# = #env0.tenv().map().get(id)#;
    # check(tsym != null, "undeclared table " + id.name()); #
    env1: #Env# = #env0.enterTable(tsym.csyms())#;
    env1 ⊢ InsertColumnClause: insertColumnClause(csyms1);
    env1 ⊢ TableExp: tableExp(csyms2);
    #
      int n1 = csyms1.list().size(); int n2 = csyms2.list().size();
      check (n1 == n2, n2 + " values given, but " + n1 + " values expected");
      for (int i = 0; i < n1; i++)
        check(csyms1.list().get(i).matchedBy(csyms2.list().get(i), false),
          "type " + csyms2.list().get(i).ctype().type() +
          " of value in column " + (i+1) + " does not match column type " +
          csyms1.list().get(i).ctype().type());
      Annotation.setTableInfo(tsym, csyms1);
    #
  }
}
judgement #Env# ⊢ InsertColumnClause: insertColumnClause(#ColumnSymbols#)
{
  inference env ⊢ NoInsertColumnClause: insertColumnClause(csyms)
  {
    csyms = #env.csyms()#;
  }
```

```
  inference env ⊢ SomeInsertColumnClause[IDs]: insertColumnClause(csyms)
  {
    ids0: #IdSeq# = #new IdSeq()#;
    ids0 ⊢ IDs: ids(ids);
    #
      int n = ids.list().size();
      for (int i1 = 0; i1 < n; i1++) {
        for (int i2 = i1+1; i2 < n; i2++) {
          check(!ids.list().get(i1).name().equals(ids.list().get(i2).name()),
            "duplicate identifier " + ids.list().get(i1).name()); } }
      List<ColumnSymbol> list = new ArrayList<ColumnSymbol>();
      for (Id id : ids.list()) {
        ColumnSymbol csym = env.cenv().map().get(id);
        check(csym != null, "undeclared column " + id.name());
        list.add(csym); }
    #
    csyms = #new ColumnSymbols(list)#;
  }
}
judgement #IdSeq# ⊢ IDs: ids(#IdSeq#)
{
  inference ids0 ⊢ SingleID[ID]: ids(ids)
  {
    id: #Id# = #new Id(ID)#;
    ids = #ids0.add(id)#;
  }
  inference ids0 ⊢ MultipleIDs[ID,IDs]: ids(ids)
  {
    id: #Id# = #new Id(ID)#;
    ids1: #IdSeq# = #ids0.add(id)#;
    ids1 ⊢ IDs: ids(ids);
  }
}
judgement #Env# ⊢ UpdateStat: updateStat
{
  inference env0 ⊢ Update[ID,Assignments,WhereClause,Annotation]: updateStat
  {
    id: #Id# = #new Id(ID)#;
    tsym: #TableSymbol# = #env0.tenv().map().get(id)#;
    #check(tsym != null, "undeclared table " + ID);#
    csyms: #ColumnSymbols# = #tsym.csyms()#;
    env1: #Env# = #env0.enterTable(csyms)#;
    env1 ⊢ WhereClause: whereClause;
    idset0: #IdSet# = #new IdSet()#;
    env2: #Env# = #env1.enterRow()#;
    env2,csyms,idset0 ⊢ Assignments: assignments(ids);
    # Annotation.setTableSymbol(tsym); #
  }
}
judgement #Env#,#ColumnSymbols#,#IdSet# ⊢ Assignments: assignments(#IdSet#)
{
  inference env,csyms,ids0 ⊢ SingleAssignment[Assignment]: assignments(ids)
  {
    env,csyms,ids0 ⊢ Assignment: assignment(ids);
```

```
  }
  inference env,csyms,ids0 ⊢ MultipleAssignments[Assignment,Assignments]:
    assignments(ids)
  {
    env,csyms,ids0 ⊢ Assignment: assignment(ids1);
    env,csyms,ids1 ⊢ Assignments: assignments(ids);
  }
}
judgement #Env#,#ColumnSymbols#,#IdSet# ⊢ Assignment: assignment(#IdSet#)
{
  inference env,csyms,ids0 ⊢ TheAssignment[ID,Exp,Annotation]: assignment(ids)
  {
    id: #Id# = #new Id(ID)#;
    # check(!ids0.set().contains(id),
        "duplicate assignment to column " + id.name()); #
    ids = #ids0.add(id)#;
    csym: #ColumnSymbol# = #env.cenv().map().get(id)#;
    # check(csym != null, "undeclared column " + id.name()); #
    env ⊢ Exp: exp(etype);
    #
      check(csym.matchedBy(etype.type()),
        "type of expression " + Exp + " does not match column type");
    #
    cpos: #Integer# = #0#;
    #
      int n = csyms.list().size();
      while (cpos < n && csyms.list().get(cpos) != csym) cpos++;
      check(cpos < n, "current table does not have column " + id.name());
      Annotation.setPosition(cpos);
    #
  }
}
judgement #Env# ⊢ DeleteStat: deleteStat
{
  inference env0 ⊢ Delete[ID,WhereClause,Annotation]: deleteStat
  {
    id: #Id# = #new Id(ID)#;
    tsym: #TableSymbol# = #env0.tenv().map().get(id)#;
    #check(tsym != null, "undeclared table " + ID);#
    csyms: #ColumnSymbols# = #tsym.csyms()#;
    env1: #Env# = #env0.enterTable(csyms)#;
    env1 ⊢ WhereClause: whereClause;
    # Annotation.setTableSymbol(tsym); #
  }
}
judgement #Env# ⊢ SelectStat: selectStat
{
  inference env ⊢ Select[TableExp,Annotation]: selectStat
  {
    env ⊢ TableExp: tableExp(csyms);
    # Annotation.setColumnSymbols(csyms); #
  }
}
judgement #Env# ⊢ TableExp: tableExp(#ColumnSymbols#)
```

```
{
  inference env ⊢ TableExpValuesExp[ValuesExp]: tableExp(csyms)
  {
    env ⊢ ValuesExp: valuesExp(csyms);
  }
  inference env ⊢ TableExpSelectExp[SelectExp]: tableExp(csyms)
  {
    env ⊢ SelectExp: selectExp(csyms);
  }
  inference env ⊢ TableExpCombination[TableExp1,SetOp,DistinctClause,TableExp2]:
    tableExp(csyms)
  {
    env ⊢ TableExp1: tableExp(csyms1);
    env ⊢ TableExp2: tableExp(csyms2);
    csyms = #csyms1.mergeWith(csyms2)#;
  }
}
judgement #Env# ⊢ ValuesExp: valuesExp(#ColumnSymbols#)
{
  inference env ⊢ TheValuesExp[RowExps]: valuesExp(csyms)
  {
    env ⊢ RowExps: rowExps(csyms);
  }
}
judgement #Env# ⊢ RowExps: rowExps(#ColumnSymbols#)
{
  inference env ⊢ SingleRowExp[RowExp]: rowExps(csyms)
  {
    env ⊢ RowExp: rowExp(csyms);
  }
  inference env ⊢ MultipleRowExps[RowExp,RowExps]: rowExps(csyms)
  {
    env ⊢ RowExp: rowExp(csyms1);
    env ⊢ RowExps: rowExps(csyms2);
    csyms = #csyms1.mergeWith(csyms2)#;
  }
}
judgement #Env# ⊢ RowExp: rowExp(#ColumnSymbols#)
{
  inference env ⊢ TheRowExp[Exps]: rowExp(csyms)
  {
    etypes0: #ExpTypes# = #new ExpTypes()#;
    env,etypes0 ⊢ Exps: exps(etypes);
    csyms = #etypes.columnSymbols()#;
  }
}
judgement #Env# ⊢ SelectExp: selectExp(#ColumnSymbols#)
{
  inference env0 ⊢ TheSelectExp[DistinctClause,SelectClause,FromClause,
    WhereClause,GroupByClause,HavingClause,OrderByClause]: selectExp(csyms)
  {
    env0 ⊢ FromClause: fromClause(env1);
    env1 ⊢ WhereClause: whereClause;
    env1 ⊢ GroupByClause: groupByClause(env2);
```

```
    env2 ⊢ HavingClause: havingClause;
    env2 ⊢ SelectClause: selectClause(env3,csyms);
    env3 ⊢ OrderByClause: orderByClause;
  }
}
judgement #Env# ⊢ FromClause: fromClause(#Env#)
{
  inference env0 ⊢ NoFromClause: fromClause(env)
  {
    env = #env0.enterTable(new ColumnSymbols())#;
  }
  inference env0 ⊢ SomeFromClause[TableD]: fromClause(env)
  {
    tableEnv: #Env# = #Env.setTableEnv(env0)#;
    cenv0: #ColumnEnv# = #env0.cenv()#;
    emptyIdSet: #IdSet# = #new IdSet()#;
    tableEnv,env0,cenv0,emptyIdSet ⊢ TableD: table(env1,cenv1,ids,csyms);
    env = #env1.enterTable(cenv1,csyms)#;
  }
}
judgement #Env#,#Env#,#ColumnEnv#,#IdSet# ⊢
  TableD: table(#Env#,#ColumnEnv#,#IdSet#,#ColumnSymbols#)
{
  inference env0,env1,cenv1,ids0 ⊢ TableId[ID,AsClause,Annotation]:
    table(env,cenv,ids,csyms)
  {
    #
      Id id = new Id(ID);
      TableSymbol tsym = env0.tenv().map().get(id);
      check(tsym != null, "undeclared table " + ID);
    #
    csyms = #tsym.csyms()#;
    ⊢ AsClause: asClause(idopt);
    # if (idopt.option().isEmpty()) { #
        ids = #ids0.add(id)#;
        env = #ids0.set().contains(id) ? env1.removeTable(id) :
          env1.addTable(id,tsym)#;
        cenv = #cenv1.addColumns(csyms)#;
    # } else {
        Id id0 = idopt.option().get();
        check(!ids0.set().contains(id0),
          "double declaration of table identifier " + id0.name());
        ids = ids0.add(id0);
        env = env1.addTable(id0,tsym);
        cenv = cenv1;
     }
     Annotation.setTableSymbol(tsym);
    #
  }
  inference env0,env1,cenv1,ids0 ⊢
    TableTableExp[TableExp,AsClause]: table(env,cenv,ids,csyms)
  {
    env0 ⊢ TableExp: tableExp(csyms);
    ⊢ AsClause: asClause(idopt);
```

```
    # if (idopt.option().isEmpty()) { #
        ids = ids0;
        env = env1;
        cenv = #cenv1.addColumns(csyms)#;
    # } else {
        Id id0 = idopt.option().get();
        check(!ids0.set().contains(id0),
          "double declaration of table identifier " + id0.name());
        TableSymbol tsym = new TableSymbol(id0, csyms);
        ids = ids0.add(id0);
        env = env1.addTable(id0,tsym);
        cenv = cenv1;
      }
    #
  }
  inference env0,env1,cenv1,ids0 ⊢
    TableJoin[TableD1,JoinClause,TableD2,JoinCondition,Annotation]:
      table(env,cenv,ids,csyms)
  {
    env0,env1,cenv1,ids0 ⊢ TableD1: table(env2,cenv2,ids1,csyms1);
    env0,env2,cenv2,ids1 ⊢ TableD2: table(env,cenv,ids,csyms2);
    env,csyms1,csyms2 ⊢ JoinCondition: joinCondition;
    csyms = #csyms1.add(csyms2)#;
    # Annotation.setColumnSymbolsPair(csyms1,csyms2); #
  }
  inference env0,env1,cenv1,ids0 ⊢
    TableCrossJoin[TableD1,CrossJoinClause,TableD2]: table(env,cenv,ids,csyms)
  {
    env0,env1,cenv1,ids0 ⊢ TableD1: table(env2,cenv2,ids1,csyms1);
    env0,env2,cenv2,ids1 ⊢ TableD2: table(env,cenv,ids,csyms2);
    csyms = #csyms1.add(csyms2)#;
  }
}
judgement ⊢ AsClause: asClause(#IdOption#)
{
  inference ⊢ NoAsClause: asClause(idOption)
  {
    idOption = #new IdOption(Optional.empty())#;
  }
  inference ⊢ SomeAsClause[ID]: asClause(idOption)
  {
    id: #Id# = #new Id(ID)#;
    idOption = #new IdOption(Optional.of(id))#;
  }
  inference ⊢ SomeAsClauseAs[ID]: asClause(idOption)
  {
    id: #Id# = #new Id(ID)#;
    idOption = #new IdOption(Optional.of(id))#;
  }
}
judgment #Env#,#ColumnSymbols#,#ColumnSymbols# ⊢ JoinCondition: joinCondition
{
  inference env,csyms1,csyms2 ⊢ JoinOn[Exp]: joinCondition
  {
```

```
    csyms: #ColumnSymbols# = #csyms1.add(csyms2)#;
    env1: #Env# = #env.enterTable(csyms).enterRow()#;
    env1 ⊢ Exp:exp(etype);
    # check(etype.type() instanceof Type.Bool, "expression " + Exp +
        " is not a Boolean but has type " + etype.type()); #
  }
  inference env,csyms1,csyms2 ⊢ JoinUsing[IDs,Annotation]: joinCondition
  {
    ids0: #IdSeq# = #new IdSeq()#;
    ids0 ⊢ IDs: ids(ids);
    #
      int n1 = csyms1.list().size();
      List<Integer> pos1 = new ArrayList<Integer>();
      List<Integer> pos2 = new ArrayList<Integer>();
      for (Id id : ids.list())
      {
        int p1 = csyms1.pos(id);
        int p2 = csyms2.pos(id);
        check(p1 != -1, "first table does not have column " + id.name());
        check(p2 != -1, "second table does not have column " + id.name());
        pos1.add(p1);
        pos2.add(p2+n1);
      }
      Annotation.setPositionSeqPair(pos1,pos2);
    #
  }
}
judgement #Env# ⊢ WhereClause: whereClause
{
  inference env ⊢ NoWhere: whereClause
  {
    // axiom (no prerequisites)
  }
  inference env ⊢ SomeWhere[Exp]: whereClause
  {
    env0: #Env# = #env.enterRow()#;
    env0 ⊢ Exp: exp(etype);
    # check(etype.matches(new Type.Bool()), "expression " + Exp +
        " is not a Boolean but has type " + etype.type()); #
  }
}
judgement #Env# ⊢ GroupByClause: groupByClause(#Env#)
{
  inference env ⊢ NoGroupBy: groupByClause(env0)
  {
    // (axiom, no prerequistes)
    csyms: #ColumnSymbols# = #new ColumnSymbols()#;
    env0 = #env.setGrouped(csyms)#;
  }
  inference env ⊢ SomeGroupBy[ColumnRefs,Annotation]: groupByClause(env0)
  {
    csyms0: #ColumnSymbols# = #new ColumnSymbols()#;
    env,csyms0 ⊢ ColumnRefs: columnRefs(csyms);
    #
```

```
      List<Integer> pos = new ArrayList<Integer>();
      for (ColumnSymbol csym : csyms.list())
      {
        Id id = csym.id();
        int p = env.csyms().pos(id);
        check(p != -1, "table does not have column " + id.name());
        pos.add(p);
      }
      Annotation.setPositionSeq(pos);
    #
    env0 = #env.setGrouped(csyms)#;
  }
}
judgement #Env# ⊢ HavingClause: havingClause
{
  inference env ⊢ NoHaving: havingClause
  {
    // (axiom, no prerequistes)
  }
  inference env ⊢ SomeHaving[Exp]: havingClause
  {
    env0: #Env# = #env.enterRow()#;
    env0 ⊢ Exp: exp(etype);
    # check(etype.matches(new Type.Bool()), "expression " + Exp +
        " is not a Boolean "); #
    # check(etype.kind() == Kind.cell, "expression " + Exp +
        " does not denote a single value but a column"); #
  }
}
judgement #Env# ⊢
  SelectClause: selectClause(#Env#,#ColumnSymbols#)
{
  inference env0 ⊢ TheSelectClause[ColumnExps,Annotation]:
  selectClause(env,csyms)
  {
    env1: #Env# = #env0.clearColumns()#;
    idset0: #IdSet# = #new IdSet()#;
    csyms0: #ColumnSymbols# = #new ColumnSymbols()#;
    env0,env1,idset0,csyms0 ⊢
      ColumnExps: columnExps(env2,ids,csyms,grouped);
    env = #env2.enterTable(csyms)#;
    # Annotation.setValue(grouped); #
  }
}
judgement #Env#,#Env#,#IdSet#,#ColumnSymbols# ⊢ ColumnExps:
  columnExps(#Env#,#IdSet#,#ColumnSymbols#,#Boolean#)
{
  inference env0,env1,idset0,csyms0 ⊢ SingleColumnExp[ColumnExp]:
    columnExps(env,idset,csyms,grouped)
  {
    env0,env1,idset0,csyms0 ⊢
      ColumnExp: columnExp(env,idset,csyms,grouped);
  }
  inference env0,env1,idset0,csyms0 ⊢
```

```
    MultipleColumnExps[ColumnExp,ColumnExps]:
      columnExps(env,idset,csyms,grouped)
  {
    env0,env1,idset0,csyms0 ⊢
      ColumnExp: columnExp(env2,idset1,csyms1,grouped1);
    env0,env2,idset1,csyms1 ⊢ ColumnExps:
      columnExps(env,idset,csyms,grouped2);
    grouped = #grouped1 && grouped2#;
  }
}
judgement #Env#,#Env#,#IdSet#,#ColumnSymbols# ⊢
  ColumnExp: columnExp(#Env#,#IdSet#,#ColumnSymbols#,#Boolean#)
{
  inference env0,env1,idset0,csyms0 ⊢
    ColumnExpAll: columnExp(env,idset,csyms,grouped)
  {
    csyms1: #ColumnSymbols# = #env0.csyms()#;
    csyms = #csyms0.add(csyms1)#;
    env = env1;
    idset = idset0;
    #
      for (ColumnSymbol csym : csyms1.list())
      {
        Id cid = csym.id();
        if (idset.set().contains(cid))
          env = env.removeColumn(cid);
        else
          env = env.addColumn(cid,csym);
        idset = idset.add(cid);
      }
    #
    grouped = #true#;
    #
      for (ColumnSymbol csym: csyms1.list())
      {
        if (!env.gsyms().contains(csym)) { grouped = false; break; }
      }
    #
  }
  inference env0,env1,idset0,csyms0 ⊢
    ColumnExpId[ID,Annotation]: columnExp(env,idset,csyms,grouped)
  {
    id: #Id# = #new Id(ID)#;
    tsym: #TableSymbol# = #env0.tenv().map().get(id)#;
    # check(tsym != null, "unknown table " + id.name()); #
    csyms1: #ColumnSymbols# = #tsym.csyms()#;
    csyms = #csyms0.add(csyms1)#;
    env = env1;
    idset = idset0;
    #
      List<Integer> pos = new ArrayList<Integer>();
      for (ColumnSymbol csym : csyms1.list())
      {
        Id cid = csym.id();
```

```
      if (idset.set().contains(cid))
        env = env.removeColumn(cid);
      else
        env = env.addColumn(cid,csym);
      idset = idset.add(cid);
      int p = env.csyms().pos(cid);
      check(p != -1, "table does not have column " + cid.name());
      pos.add(p);
    }
    Annotation.setPositionSeq(pos);
  #
  grouped = #true#;
  #
    for (ColumnSymbol csym: csyms1.list())
    {
      if (!env.gsyms().contains(csym)) { grouped = false; break; }
    }
  #
}
inference env0,env1,idset0,csyms0 ⊢
  ColumnExpExp[Exp,AsClause]: columnExp(env,idset,csyms,grouped)
{
  env2: #Env# = #env0.enterRow()#;
  env2 ⊢ Exp: exp(etype);
  ⊢ AsClause: asClause(idopt);
  grouped = #null#;
  #
    ColumnSymbol csym0 = null;
    if (Exp instanceof Exp.ExpRef)
    {
      Annotation annotation = ((Exp.ExpRef)Exp)._2();
      csym0 = annotation.getColumnSymbol();
      grouped = env2.gsyms().contains(csym0);
    }
    else
      grouped = etype.grouped();
    Id id;
    ColumnSymbol csym;
    if (idopt.option().isPresent())
    {
      id = idopt.option().get();
      check(!idset0.set().contains(id),
        "double declaration of column identifier " + id.name());
      csym = new ColumnSymbol(id,etype.type().columnType(),
        new ConstraintSet(), new IdOption(Optional.empty()));
    }
    else if (!(Exp instanceof Exp.ExpRef))
    {
      id = new Id("_");
      csym = new ColumnSymbol(id,etype.type().columnType(),
        new ConstraintSet(), new IdOption(Optional.empty()));
    }
    else
    {
```

```
        csym = csym0;
        id = csym.id();
      }
    #
    idset = #idset0.add(id)#;
    env = #idset.set().contains(id)?
      env1.removeColumn(id) : env1.addColumn(id,csym)#;
    csyms = #csyms0.add(csym)#;
  }
}
judgement #Env# ⊢ OrderByClause: orderByClause
{
  inference env ⊢ OrderByNone: orderByClause
  {
    // (axiom, no prerequisites)
  }
  inference env ⊢ OrderBySome[OrderByColumns,Annotation]: orderByClause
  {
    csymset0: #ColumnSymbolSet# = #new ColumnSymbolSet()#;
    env,csymset0 ⊢ OrderByColumns: orderByColumns(csymset);
    #
      Annotation annot = switch (OrderByColumns)
        { case OrderByColumns.SingleOrderByColumn(var c,var a) -> a;
          case OrderByColumns.MultipleOrderByColumns(var c,var cs,var a) -> a;
        };
      List<Integer> pos = annot.getPosSeq();
      List<Boolean> asc = annot.getAscSeq();
      Annotation.setPosAscSeq(pos,asc);
    #
  }
}
judgement #Env#,#ColumnSymbolSet# ⊢
  OrderByColumns:orderByColumns(#ColumnSymbolSet#)
{
  inference env,csymset0 ⊢ SingleOrderByColumn[OrderByColumn,Annotation]:
    orderByColumns(csymset)
  {
    env,csymset0 ⊢ OrderByColumn: orderByColumn(csymset);
    #
      Annotation annot = switch (OrderByColumn)
        { case OrderByColumn.TheOrderByColumn(var c,var o,var a) -> a; };
      List<Integer> pos = new ArrayList<Integer>();
      List<Boolean> asc = new ArrayList<Boolean>();
      pos.add(annot.getPos());
      asc.add(annot.getAsc());
      Annotation.setPosAscSeq(pos,asc);
    #
  }
  inference env,csymset0 ⊢
    MultipleOrderByColumns[OrderByColumn,OrderByColumns,Annotation]:
    orderByColumns(csymset)
  {
    env,csymset0 ⊢ OrderByColumn: orderByColumn(csymset1);
    env,csymset1 ⊢ OrderByColumns: orderByColumns(csymset);
```

```
    #
      Annotation annot1 = switch (OrderByColumn)
        { case OrderByColumn.TheOrderByColumn(var c,var o,var a) -> a; };
      Annotation annot2 = switch (OrderByColumns)
        { case OrderByColumns.SingleOrderByColumn(var c,var a) -> a;
          case OrderByColumns.MultipleOrderByColumns(var c,var cs,var a) -> a;
        };
      List<Integer> pos = new ArrayList<Integer>(annot1.getPosSeq());
      List<Boolean> asc = new ArrayList<Boolean>(annot1.getAscSeq());
      pos.addAll(annot2.getPosSeq());
      asc.addAll(annot2.getAscSeq());
      Annotation.setPosAscSeq(pos,asc);
    #
  }
}
judgement #Env#,#ColumnSymbolSet# ⊢ OrderByColumn:
  orderByColumn(#ColumnSymbolSet#)
{
  inference env,csymset0 ⊢ TheOrderByColumn[ColumnRef,OrderClause,Annotation]:
    orderByColumn(csymset)
  {
    env ⊢ ColumnRef: columnRef(csym);
    # check(!csymset0.set().contains(csym), "duplicate order column " + ColumnRef); #
    csymset = #csymset0.add(csym)#;
    p: #Integer# = #env.csyms().list().indexOf(csym)#;
    # check(p != -1, "ORDER column " + ColumnRef + " is not in SELECT clause"); #
    ⊢ OrderClause: orderClause(a);
    # Annotation.setPosAsc(p,a); #
  }
}
judgement ⊢ OrderClause: orderClause(#Boolean#)
{
  inference ⊢ OrderNone: orderClause(a)
  {
    a = #true#;
  }
  inference ⊢ OrderAsc: orderClause(a)
  {
    a = #true#;
  }
  inference ⊢ OrderDesc: orderClause(a)
  {
    a = #false#;
  }
}
judgement #Env#,#ExpTypes# ⊢ Exps: exps(#ExpTypes#)
{
  inference env,etypes0 ⊢ SingleExp[Exp]: exps(etypes)
  {
    env ⊢ Exp: exp(etype);
    etypes = #etypes0.add(etype)#;
  }
  inference env,etypes0 ⊢ MultipleExps[Exp,Exps]: exps(etypes)
  {
```

```
    env ⊢ Exp: exp(etype);
    etypes1: #ExpTypes# = #etypes0.add(etype)#;
    env,etypes1 ⊢ Exps: exps(etypes);
  }
}
judgement #Env# ⊢ Exp: exp(#ExpType#)
{
  inference env ⊢ ExpMetaVar[ID]: exp(etype)
  {
    id: #Id# = #new Id(ID)#;
    vsym: #VarSymbol# = #env.venv().map().get(id)#;
    # check(vsym != null, "undeclared meta-variable " + ID); #
    type: #Type# =
      #switch (vsym.type()) {
         case intvar -> new Type.Int();
         case boolvar -> new Type.Bool();
         case stringvar -> new Type.String();
       }#;
    etype = #new ExpType(type,Kind.cell,true)#;
  }
  inference env ⊢ ExpLiteral[Literal,Annotation]: exp(etype)
  {
    ⊢ Literal: literal(type,value);
    etype = #new ExpType(type,Kind.cell,true)#;
    # Annotation.setValue(value); #
  }
  inference env ⊢ ExpRef[ColumnRef,Annotation]: exp(etype)
  {
    env ⊢ ColumnRef: columnRef(csym);
    kind: #Kind# = #env.ckind().get(csym)#;
    depth: #Integer# = #env.cdepth().get(csym)#;
    pos: #Integer# = #env.cstack().get(depth).list().indexOf(csym)#;
    type: #Type# = #csym.ctype().type()#;
    # boolean grouped = depth < env.cstack().size()-1 ||
        env.gsyms().contains(csym); #
    etype = #new ExpType(type, kind, grouped)#;
    # Annotation.setColumnInfo(csym,pos,kind,depth); #
  }
  inference env ⊢ ExpUnary[UnaryOp,Exp]: exp(etype)
  {
    env ⊢ Exp: exp(etype0);
    etype0,Exp ⊢ UnaryOp: unaryOp(etype);
  }
  inference env ⊢ ExpBinary[Exp1,BinaryOp,Exp2]: exp(etype)
  {
    env ⊢ Exp1: exp(etype1);
    env ⊢ Exp2: exp(etype2);
    etype1,etype2,Exp1,Exp2 ⊢ BinaryOp: binaryOp(etype);
  }
  inference env ⊢ ExpExps[Exps]: exp(etype)
  {
    etypes0: #ExpTypes# = #new ExpTypes()#;
    env,etypes0 ⊢ Exps: exps(etypes);
  # if (etypes.list().size() == 1) #
```

```
    etype = #etypes.list().get(0)#;
# else {
    List<Type> types = new ArrayList<Type>();
    Kind kind = null;
    boolean grouped = true;
    for (ExpType etype0 : etypes.list())
    {
      types.add(etype0.type());
      kind = Kind.combine(kind, etype0.kind());
      grouped = grouped && etype0.grouped();
    }
    Type type = new Type.Seq(types);
    etype = new ExpType(type,kind,grouped);
  } #
}
inference env ⊢ ExpSelect[SelectExp]: exp(etype)
{
  env ⊢ SelectExp: selectExp(csyms);
  # List<Type> types = new ArrayList<Type>(); #
  # for (ColumnSymbol csym : csyms.list()) types.add(csym.ctype().type()); #
  etype = #types.size() == 1 ?
    new ExpType(types.get(0),Kind.column,false) :
    new ExpType(new Type.Seq(types),Kind.column,false)# ;
}
inference env ⊢ ExpBetween[Exp1,NotClause,Exp2,Exp3]: exp(etype)
{
  env ⊢ Exp1: exp(etype1);
  env ⊢ Exp2: exp(etype2);
  env ⊢ Exp3: exp(etype3);
  # check(matchingAtomicTypes(etype1,etype2,etype3),
      "arguments " + Exp1 + "," + Exp2 + "," + Exp3 +
      " of operation BETWEEN have incompatible types " +
      etype1 + "," + etype2 + "," + etype3); #
  # Kind kind = Kind.combine(etype1.kind(),etype2.kind(),etype3.kind()); #
  # boolean grouped = etype1.grouped() && etype2.grouped() && etype3.grouped(); #
  etype = #new ExpType(new Type.Bool(),kind,grouped)#;
}
inference env ⊢ ExpIsNull[Exp,NotClause]: exp(etype)
{
  env ⊢ Exp: exp(etype0);
  # check(etype0.atomicType(), "argument " + Exp +
    " of operation IS NULL has non-atomic type " + etype0.type()); #
  etype = #new ExpType(new Type.Bool(),etype0.kind(),etype0.grouped())#;
}
inference env ⊢ ExpIsTrue[Exp,NotClause]: exp(etype)
{
  env ⊢ Exp: exp(etype0);
  # check(etype0.matches(new Type.Bool()), "argument " + Exp +
    " of operation IS TRUE is not a Boolean but has type " + etype0.type()); #
  etype = #new ExpType(new Type.Bool(),etype0.kind(),etype0.grouped())#;
}
inference env ⊢ ExpIsFalse[Exp,NotClause]: exp(etype)
{
  env ⊢ Exp: exp(etype0);
```

```
  # check(etype0.matches(new Type.Bool()), "argument " + Exp +
    " of operation IS FALSE is not a Boolean but has type " + etype0.type()); #
  etype = #new ExpType(new Type.Bool(),etype0.kind(),etype0.grouped())#;
}
inference env ⊢ ExpIsDistinct[Exp1,NotClause,Exp2]: exp(etype)
{
  env ⊢ Exp1: exp(etype1);
  env ⊢ Exp2: exp(etype2);
  # check(matchingAtomicTypes(etype1,etype2),
      "arguments " + Exp1 + "," + Exp2 +
      " of operation IS DISTINCT have incompatible types " +
      etype1 + "," + etype2); #
  # Kind kind = Kind.combine(etype1.kind(),etype2.kind()); #
  # boolean grouped = etype1.grouped() && etype2.grouped(); #
  etype = #new ExpType(new Type.Bool(),kind,grouped)#;
}
inference env ⊢ ExpLike[Exp,STR,Annotation]: exp(etype)
{
  env ⊢ Exp: exp(etype);
  # check(etype.matches(new Type.String()), "argument " + Exp +
    " of operation LIKE is not a string but has type " + etype.type()); #
  etype = #new ExpType(new Type.Bool(),etype.kind(),etype.grouped())#;
  # Annotation.setPattern(STR.substring(1,STR.length()-1)); #
}
inference env ⊢ ExpCase[WhenClauses,ElseClause]: exp(etype)
{
  etype0: #ExpType# = #new ExpType(new Type.Null(), Kind.cell, true)#;
  env,etype0 ⊢ WhenClauses: whenClauses(etype1);
  env,etype1 ⊢ ElseClause: elseClause(etype);
}
inference env ⊢ ExpCaseExp[Exp,WhenClauses,ElseClause]: exp(etype)
{
  etype0: #ExpType# = #new ExpType(new Type.Null(), Kind.cell, true)#;
  env ⊢ Exp: exp(etype1);
  env,etype0,etype1 ⊢ WhenClauses: whenClausesExp(etype2);
  env,etype2 ⊢ ElseClause: elseClause(etype);
}
inference env ⊢ ExpNullIf[Exp1,Exp2]: exp(etype)
{
  env ⊢ Exp1: exp(etype1);
  env ⊢ Exp2: exp(etype2);
  # check(matchingAtomicTypes(etype1,etype2),
      "arguments " + Exp1 + "," + Exp2 +
      " of operation NULLIF have incompatible types " +
      etype1 + "," + etype2); #
  # Kind kind = Kind.combine(etype1.kind(),etype2.kind()); #
  # boolean grouped = etype1.grouped() && etype2.grouped(); #
  etype = #new ExpType(etype1.type(),kind,grouped)#;
}
inference env ⊢ ExpCoalesce[Exps]: exp(etype)
{
  etypes0: #ExpTypes# = #new ExpTypes()#;
  env,etypes0 ⊢ Exps: exps(etypes);
  #
```

```
    ExpType[] etypes1 =
      etypes.list().toArray(new ExpType[etypes.list().size()]);
    check(matchingAtomicTypes(etypes1),
        "arguments of COALESCE have incompatible types");
  #
  etype = #ExpType.combine(etypes1)#;
}
inference env ⊢ ExpExists[SelectExp]: exp(etype)
{
  env ⊢ SelectExp: selectExp(csyms);
  etype = #new ExpType(new Type.Bool(), Kind.cell, true)#;
}
inference env ⊢ ExpIn[Exp,NotClause,Exps]: exp(etype)
{
  env ⊢ Exp: exp(etype0);
  etypes0: #ExpTypes# = #new ExpTypes()#;
  env,etypes0 ⊢ Exps: exps(etypes);
  #
    ExpTypes etypes1 = etypes.add(etype0);
    ExpType[] etypes2 = etypes1.list().toArray(new ExpType[etypes1.list().size()]);
    check(matchingAtomicTypes(etypes2),
      "arguments of operation IN have types that are incompatible with type " +
      etype0 + " of expression " + Exp);
    Kind kind = null;
    for (ExpType etype2 : etypes2) kind = Kind.combine(kind,etype2.kind());
    boolean grouped = true;
    for (ExpType etype2 : etypes2) grouped = grouped && etype2.grouped();
  #
  etype = #new ExpType(new Type.Bool(), kind, grouped)#;
}
inference env ⊢ ExpInSelect[Exp,NotClause,SelectExp]: exp(etype)
{
  env ⊢ Exp: exp(etype0);
  env ⊢ SelectExp: selectExp(csyms);
  #
    check(csyms.list().size() == 1, "argument " + SelectExp +
      " of operation IN does not select one column but " +
      csyms.list().size() + " columns");
    Type type1 = etype0.type();
    Type type2 = csyms.list().get(0).ctype().type();
    check(matchingAtomicTypes(type1,type2), "argument " + Exp +
      " of operation IN has type " + type1 +
      " which is incompatible with type " + type2 +
      " of argument " + SelectExp);
  #
  etype = #new ExpType(new Type.Bool(), etype0.kind(), etype0.grouped())#;
}
inference env ⊢ ExpQuantified[Exp,BinaryOp,Quantifier,SelectExp]: exp(etype)
{
  env ⊢ Exp: exp(etype0);
  env ⊢ SelectExp: selectExp(csyms);
  #
    check(csyms.list().size() == 1, "argument " + SelectExp +
      " of operation " + BinaryOp + " " + Quantifier +
```

```
        " does not select one column but " + csyms.list().size() + " columns");
    #
    etype1: #ExpType# =
      #new ExpType(csyms.list().get(0).ctype().type(), Kind.cell, true)#;
    Exp0: #Exp# = #new Exp.ExpRef(
      new ColumnRef.PlainRef(csyms.list().get(1).id().name()),
      new Annotation())#;
    etype0,etype1,Exp,Exp0 ⊢ BinaryOp: binaryOp(etype);
  }
  inference env ⊢ ExpAggregateExp[AggregateExp]: exp(etype)
  {
    env ⊢ AggregateExp: aggregateExp(type);
    etype = #new ExpType(type,Kind.cell,true)#;
  }
}
judgement #Env# ⊢ ColumnRef: columnRef(#ColumnSymbol#)
{
  inference env ⊢ PlainRef[ID]: columnRef(csym)
  {
    id: #Id# = #new Id(ID)#;
    csym = #env.cenv().map().get(id)#;
    # check(csym != null, "undeclared column " + ID); #
  }
  inference env ⊢ TableRef[ID1,ID2]: columnRef(csym)
  {
    tid: #Id# = #new Id(ID1)#;
    cid: #Id# = #new Id(ID2)#;
    tsym: #TableSymbol# = #env.tenv().map().get(tid)#;
    # check(tsym != null, "undeclared table " + ID1); #
    # int cpos = tsym.csyms().pos(cid); #
    # check(cpos != -1, "table " + ID1 + " does not have column " + ID2); #
    csym = #tsym.csyms().list().get(cpos)#;
  }
}
judgement #Env#,#ColumnSymbols# ⊢ ColumnRefs: columnRefs(#ColumnSymbols#)
{
  inference env,csyms0 ⊢ SingleColumnRef[cref]: columnRefs(csyms)
  {
    env ⊢ cref: columnRef(csym);
    csyms = #csyms0.add(csym)#;
  }
  inference env,csyms0 ⊢ MultipleColumnRefs[cref,crefs]: columnRefs(csyms)
  {
    env ⊢ cref: columnRef(csym);
    csyms1: #ColumnSymbols# = #csyms0.add(csym)#;
    env,csyms1 ⊢ crefs: columnRefs(csyms);
  }
}
judgement ⊢ Literal: literal(#Type#,#Value#)
{
  inference ⊢ LiteralNUM[NUM]: literal(type,value)
  {
    type = #new Type.Int()#;
    value = #new Value.Int(Integer.parseInt(NUM))#;
```

```
  }
  inference ⊢ LiteralSTR[STR]: literal(type,value)
  {
    type = #new Type.String()#;
    value = #new Value.String(STR.substring(1, STR.length()-1))#;
  }
  inference ⊢ LiteralNull: literal(type,value)
  {
    type = #new Type.Null()#;
    value = #new Value.Null()#;
  }
  inference ⊢ LiteralTrue: literal(type,value)
  {
    type = #new Type.Bool()#;
    value = #new Value.Bool(true)#;
  }
  inference ⊢ LiteralFalse: literal(type,value)
  {
    type = #new Type.Bool()#;
    value = #new Value.Bool(false)#;
  }
}
judgement #ExpType#,#Exp# ⊢ UnaryOp: unaryOp(#ExpType#)
{
  inference etype0,exp0 ⊢ UnaryNot: unaryOp(etype)
  {
    # Type btype = new Type.Bool(); #
    # check(etype0.matches(btype), "argument " + exp0 +
      " of operation NOT is not a Boolean but has type " + etype0.type()); #
    etype = #new ExpType(btype, etype0.kind(), etype0.grouped())#;
  }
  inference etype0,exp0 ⊢ UnaryMinus: unaryOp(etype)
  {
    # Type itype = new Type.Int(); #
    # check(etype0.matches(itype), "argument " + exp0 +
      " of operation - is not an integer but has type " + etype0.type()); #
    etype = #new ExpType(itype, etype0.kind(), etype0.grouped())#;
  }
}
judgement #ExpType#,#ExpType#,#Exp#,#Exp# ⊢ BinaryOp: binaryOp(#ExpType#)
{
  inference etype1,etype2,exp1,exp2 ⊢ BinaryPlus: binaryOp(etype)
  {
    # Type itype = new Type.Int(); #
    # check(etype1.matches(itype), "first argument " + exp1 +
      " of operation + is not an integer but has type " + etype1.type()); #
    # check(etype2.matches(itype), "second argument " + exp2 +
      " of operation + is not an integer but has type " + etype2.type()); #
    etype = #itype.expType(etype1,etype2)#;
  }
  inference etype1,etype2,exp1,exp2 ⊢ BinaryMinus: binaryOp(etype)
  {
    # Type itype = new Type.Int(); #
    # check(etype1.matches(itype), "first argument " + exp1 +
```

```
    " of operation - is not an integer but has type " + etype1.type()); #
  # check(etype2.matches(itype), "second argument " + exp2 +
    " of operation - is not an integer but has type " + etype2.type()); #
  etype = #itype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryTimes: binaryOp(etype)
{
  # Type itype = new Type.Int(); #
  # check(etype1.matches(itype), "first argument " + exp1 +
    " of operation * is not an integer but has type " + etype1.type()); #
  # check(etype2.matches(itype), "second argument " + exp2 +
    " of operation * is not an integer but has type " + etype2.type()); #
  etype = #itype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryDiv: binaryOp(etype)
{
  # Type itype = new Type.Int(); #
  # check(etype1.matches(itype), "first argument " + exp1 +
    " of operation / is not an integer but has type " + etype1.type()); #
  # check(etype2.matches(itype), "second argument " + exp2 +
    " of operation / is not an integer but has type " + etype2.type()); #
  etype = #itype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryMod: binaryOp(etype)
{
  # Type itype = new Type.Int(); #
  # check(etype1.matches(itype), "first argument " + exp1 +
    " of operation % is not an integer but has type " + etype1.type()); #
  # check(etype2.matches(itype), "second argument " + exp2 +
    " of operation % is not an integer but has type " + etype2.type()); #
  etype = #itype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryBitAnd: binaryOp(etype)
{
  # Type itype = new Type.Int(); #
  # check(etype1.matches(itype), "first argument " + exp1 +
    " of operation & is not an integer but has type " + etype1.type()); #
  # check(etype2.matches(itype), "second argument " + exp2 +
    " of operation & is not an integer but has type " + etype2.type()); #
  etype = #itype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryBitOr: binaryOp(etype)
{
  # Type itype = new Type.Int(); #
  # check(etype1.matches(itype), "first argument " + exp1 +
    " of operation | is not an integer but has type " + etype1.type()); #
  # check(etype2.matches(itype), "second argument " + exp2 +
    " of operation | is not an integer but has type " + etype2.type()); #
  etype = #itype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryBitExclOr: binaryOp(etype)
{
  # Type itype = new Type.Int(); #
  # check(etype1.matches(itype), "first argument " + exp1 +
```

```
    " of operation ^ is not an integer but has type " + etype1.type()); #
  # check(etype2.matches(itype), "second argument " + exp2 +
    " of operation ^ is not an integer but has type " + etype2.type()); #
  etype = #itype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryEqual: binaryOp(etype)
{
  # Type btype = new Type.Bool(); #
  # check(matchingAtomicTypes(etype1, etype2), "arguments " + exp1 +
    " and " + exp2 + " of operation = have incompatible types " +
    etype1.type() + " and " + etype2.type()); #
  etype = #btype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryGreater: binaryOp(etype)
{
  # Type btype = new Type.Bool(); #
  # check(matchingAtomicTypes(etype1, etype2), "arguments " + exp1 +
    " and " + exp2 + " of operation > have incompatible types " +
    etype1.type() + " and " + etype2.type()); #
  etype = #btype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryGreaterEqual: binaryOp(etype)
{
  # Type btype = new Type.Bool(); #
  # check(matchingAtomicTypes(etype1, etype2), "arguments " + exp1 +
    " and " + exp2 + " of operation >= have incompatible types " +
    etype1.type() + " and " + etype2.type()); #
  etype = #btype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryLess: binaryOp(etype)
{
  # Type btype = new Type.Bool(); #
  # check(matchingAtomicTypes(etype1, etype2), "arguments " + exp1 +
    " and " + exp2 + " of operation < have incompatible types" +
    etype1.type() + " and " + etype2.type()); #
  etype = #btype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryLessEqual: binaryOp(etype)
{
  # Type btype = new Type.Bool(); #
  # check(matchingAtomicTypes(etype1, etype2), "arguments " + exp1 +
    " and " + exp2 + " of operation <= have incompatible types " +
    etype1.type() + " and " + etype2.type()); #
  etype = #btype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryNotEqual: binaryOp(etype)
{
  # Type btype = new Type.Bool(); #
  # check(matchingAtomicTypes(etype1, etype2), "arguments " + exp1 +
    " and " + exp2 + " of operation <> have incompatible types " +
    etype1.type() + " and " + etype2.type()); #
  etype = #btype.expType(etype1,etype2)#;
}
inference etype1,etype2,exp1,exp2 ⊢ BinaryAnd: binaryOp(etype)
```

```
  {
    # Type btype = new Type.Bool(); #
    # check(etype1.matches(btype), "first argument " + exp1 +
      " of operation AND is not a Boolean but has type " + etype1.type()); #
    # check(etype2.matches(btype), "second argument " + exp2 +
      " of operation AND is not a Boolean but has type " + etype2.type()); #
    etype = #btype.expType(etype1,etype2)#;
  }
  inference etype1,etype2,exp1,exp2 ⊢ BinaryOr: binaryOp(etype)
  {
    # Type btype = new Type.Bool(); #
    # check(etype1.matches(btype), "first argument " + exp1 +
      " of operation OR is not a Boolean but has type " + etype1.type()); #
    # check(etype2.matches(btype), "second argument " + exp2 +
      " of operation OR is not a Boolean but has type " + etype2.type()); #
    etype = #btype.expType(etype1,etype2)#;
  }
}
judgement #Env#,#ExpType# ⊢ WhenClauses: whenClauses(#ExpType#)
{
  inference env,etype0 ⊢ SingleWhen[WhenClause]: whenClauses(etype)
  {
    env,etype0 ⊢ WhenClause: whenClause(etype);
  }
  inference env,etype0 ⊢ MultipleWhens[WhenClause,WhenClauses]:
    whenClauses(etype)
  {
    env,etype0 ⊢ WhenClause: whenClause(etype1);
    env,etype1 ⊢ WhenClauses: whenClauses(etype);
  }
}
judgement #Env#,#ExpType# ⊢ WhenClause: whenClause(#ExpType#)
{
  inference env,etype0 ⊢ TheWhenClause[Exp1,Exp2]: whenClause(etype)
  {
    env ⊢ Exp1: exp(etype1);
    env ⊢ Exp2: exp(etype2);
    # check(etype1.matches(new Type.Bool()), "expression " + Exp1 +
        " in WHEN clause is not a Boolean but has type " + etype1.type()); #
    # check(matchingAtomicTypes(etype0,etype2), "expression " + Exp2 +
        " in THEN clause has type " + etype2.type() +
        " that is incompatible with type " + etype0.type() +
        " of the expressions in the previous THEN clauses"); #
    #
      Type type = Type.combine(etype0.type(),etype2.type());
      Kind kind = Kind.combine(etype0.kind(),etype1.kind(),etype2.kind());
      boolean grouped = etype0.grouped() && etype1.grouped() && etype2.grouped();
    #
    etype = #new ExpType(type,kind,grouped)#;
  }
}
judgement #Env#,#ExpType#,#ExpType# ⊢ WhenClauses: whenClausesExp(#ExpType#)
{
  inference env,etype0,etype3 ⊢ SingleWhen[WhenClause]: whenClausesExp(etype)
```

```
  {
    env,etype0,etype3 ⊢ WhenClause: whenClauseExp(etype);
  }
  inference env,etype0,etype3 ⊢ MultipleWhens[WhenClause,WhenClauses]:
    whenClausesExp(etype)
  {
    env,etype0,etype3 ⊢ WhenClause: whenClauseExp(etype1);
    env,etype1,etype3 ⊢ WhenClauses: whenClausesExp(etype);
  }
}
judgement #Env#,#ExpType#,#ExpType# ⊢ WhenClause: whenClauseExp(#ExpType#)
{
  inference env,etype0,etype3 ⊢ TheWhenClause[Exp1,Exp2]: whenClauseExp(etype)
  {
    env ⊢ Exp1: exp(etype1);
    env ⊢ Exp2: exp(etype2);
    # check(matchingAtomicTypes(etype1,etype3), "expression " + Exp1 +
        " in WHEN clause has type " + etype1.type() +
        " which is incompatible with type " + etype3.type() +
        " of CASE expression"); #
    # check(matchingAtomicTypes(etype0,etype2), "expression " + Exp2 +
        " in THEN clause has type " + etype2.type() +
        " that is incompatible with type " + etype0.type() +
        " of the expressions in the previous THEN clauses"); #
    #
      Type type = Type.combine(etype0.type(),etype2.type());
      Kind kind = Kind.combine(etype0.kind(),etype1.kind(),etype2.kind());
      boolean grouped = etype0.grouped() && etype1.grouped() && etype2.grouped();
    #
    etype = #new ExpType(type,kind,grouped)#;
  }
}
judgement #Env#,#ExpType# ⊢ ElseClause: elseClause(#ExpType#)
{
  inference env,etype0 ⊢ NoElse: elseClause(etype)
  {
    etype = etype0;
  }
  inference env,etype0 ⊢ SomeElse[Exp]: elseClause(etype)
  {
    env ⊢ Exp: exp(etype1);
    # check(matchingAtomicTypes(etype0,etype1), "expression " + Exp +
        " in ELSE clause has type " + etype1.type() +
        " that is incompatible with type " + etype0.type() +
        " of the expressions in the THEN clauses"); #
    etype = #ExpType.combine(etype0,etype1)#;
  }
}
judgement #Env# ⊢ AggregateExp: aggregateExp(#Type#)
{
  inference env ⊢ AggregateExpFun[AggregateFun,AggregateArg]:
    aggregateExp(type)
  {
    env0: #Env# = #env.enterTable()#;
```

```
    env0 ⊢ AggregateArg: aggregateArg(etype);
    type0: #Type# = #etype.type()#;
    type0,AggregateArg ⊢ AggregateFun: aggregateFun(type);
  }
  inference env ⊢ AggregateExpCount: aggregateExp(type)
  {
    type = #new Type.Int()#;
  }
}
judgement #Env# ⊢ AggregateArg: aggregateArg(#ExpType#)
{
  inference env ⊢ TheAggregateArg[DistinctClause,Exp]: aggregateArg(etype)
  {
    env ⊢ Exp: exp(etype);
    # check(etype.kind() == Kind.column, "argument " + Exp +
        "of aggregate operation denotes a single value"); #
  }
}
judgement #Type#,#AggregateArg# ⊢ AggregateFun: aggregateFun(#Type#)
{
  inference type0,arg ⊢ AggregateMin: aggregateFun(type)
  {
    # check(type0 instanceof Type.Int || type0 instanceof Type.String,
        "argument " + arg +
        " of operation MIN is not an integer or a string" +
        " but has type " + type0); #
    type = type0;
  }
  inference type0,arg ⊢ AggregateMax: aggregateFun(type)
  {
    # check(type0 instanceof Type.Int || type0 instanceof Type.String,
        "argument " + arg +
        " of operation MAX is not an integer or a string" +
        " but has type " + type0); #
    type = type0;
  }
  inference type0,arg ⊢ AggregateSum: aggregateFun(type)
  {
    # check(type0 instanceof Type.Int,
        "argument " + arg + " of operation SUM is not an integer" +
        " but has type " + type0); #
    type = type0;
  }
  inference type0,arg ⊢ AggregateAvg: aggregateFun(type)
  {
    # check(type0 instanceof Type.Int,
        "argument " + arg + " of operation AVG is not an integer" +
        " but has type " + type0); #
    type = type0;
  }
  inference type0,arg ⊢ AggregateCount: aggregateFun(type)
  {
    type = #new Type.Int()#;
  }
```

```
}

// ------------------------------------------------------------------------
// denotational semantics
// ------------------------------------------------------------------------
function ⟦Session⟧: #State# → #State# × #Tables#
{
  equation ⟦SessionStats[Stats]⟧(s) = s0,tables
  {
    s0,tables = ⟦Stats⟧(s);
  }
}
function ⟦Stats⟧: #State# → #State# × #Tables#
{
  equation ⟦SingleStat[Stat]⟧(s) = s0,tables
  {
    s0,tables = ⟦Stat⟧(s);
  }
  before
  {#
    if (s.out() != null)
    {
      Annotation.show(false);
      s.out().println(Stat);
      Annotation.show(true);
      // s.out().println("> " + Stat);
    }
  #}
  equation ⟦MultipleStats[Stat,Stats]⟧(s) = s0,tables
  {
     s1,tables1 = ⟦Stat⟧(s);
     s0,tables2 = ⟦Stats⟧(s1);
     tables = #tables1.appendAll(tables2)#;
  }
  before
  {#
    if (s.out() != null)
    {
      Annotation.show(false);
      s.out().println(Stat);
      Annotation.show(true);
      // s.out().println("> " + Stat);
    }
  #}
}
function ⟦Stat⟧: #State# → #State# × #Tables#
{
  equation ⟦StatCreateTable[CreateTableStat]⟧(s) = s0,tables
  {
    s0 = ⟦CreateTableStat⟧(s);
    tables = #new Tables()#;
  }
  equation ⟦StatDropTable[DropTableStat]⟧(s) = s0,tables
  {
```

```
    s0 = ⟦DropTableStat⟧(s);
    tables = #new Tables()#;
  }
  equation ⟦StatInsert[InsertStat]⟧(s) = s0,tables
  {
    s0 = ⟦InsertStat⟧(s);
    tables = #new Tables()#;
  }
  equation ⟦StatUpdate[UpdateStat]⟧(s) = s0,tables
  {
    s0 = ⟦UpdateStat⟧(s);
    tables = #new Tables()#;
  }
  equation ⟦StatDelete[DeleteStat]⟧(s) = s0,tables
  {
    s0 = ⟦DeleteStat⟧(s);
    tables = #new Tables()#;
  }
  equation ⟦StatSelect[SelectStat]⟧(s) = s0,tables
  {
    s0 = s;
    table = ⟦SelectStat⟧(s);
    tables = #new Tables()#;
    # tables = tables.append(table); #
  }
}
function ⟦CreateTableStat⟧: #State# → #State#
{
  equation ⟦CreateTable[ID,ColumnDecls,Annotation]⟧(s) = s0
  {
    # ColumnSymbols csyms = Annotation.getColumnSymbols(); #
    # subsql.Semantics.check(s.db().getInfo(ID) == null, #
    #   "double declaration of table " + ID); #
    # s.db().createTable(ID, DatabaseUtils.info(csyms)); #
    s0 = s; // single state is passed through statements
  }
}
function ⟦DropTableStat⟧: #State# → #State#
{
  equation ⟦DropTable[ID]⟧(s) = s0
  {
    # s.db().dropTable(ID); #
    s0 = s; // single state is passed through statements
  }
}
function ⟦InsertStat⟧: #State# → #State#
{
  equation ⟦Insert[ID,InsertColumnClause,TableExp,Annotation]⟧(s) = s0
  {
    #
      TableSymbol tsym = Annotation.getTableInfo();
      ColumnSymbols csyms = Annotation.getTableColumns();
      String name = tsym.id().name();
      Database.Content content = s.db().getContent(name);
```

```
      Table table = DatabaseUtils.table(content);
    #
    s1: #State# = #s.newTable(table)#;
    rows1 = ⟦TableExp⟧(s1);
    rows2: #Table# = #rows1.tableRows(tsym, csyms)#;
    rows3: #Table# = #table.append(rows2)#;
    #
      String error = rows3.constraintViolation(tsym);
      subsql.Semantics.check(error == null, error);
      s.db().setContent(name, DatabaseUtils.content(rows3));
    #
    s0 = s; // single state is passed through statements
  }
}
function ⟦UpdateStat⟧: #State# → #State#
{
  equation ⟦Update[ID,Assignments,WhereClause,Annotation]⟧(s) = s0
  {
    #
      TableSymbol tsym = Annotation.getTableSymbol();
      String name = tsym.id().name();
      Database.Content content = s.db().getContent(name);
      Table table = DatabaseUtils.table(content);
    #
      s1: #State# = #s.newTable(table)#;
      pred = Pred⟦WhereClause⟧(s1);
    #
      List<Row> rows = new ArrayList<Row>();
      for (Row row : table.list())
      {
        if (pred.test(row))
        {
    #
          s2: #State# = #s1.newRow(row)#;
          row0: #Row# = #row#;
          row1 = ⟦Assignments⟧(s2,row0);
   #
         rows.add(row1);
        }
        else
          rows.add(row);
      }
      Table table0 = new Table(rows);
      String error = table0.constraintViolation(tsym);
      subsql.Semantics.check(error == null, error);
      s.db().setContent(name, DatabaseUtils.content(table0));
    #
    s0 = s; // single state is passed through statements
  }
}
function ⟦Assignments⟧: #State# × #Row# → #Row#
{
  equation ⟦SingleAssignment[Assignment]⟧(s,row) = row0
  {
```

```
    row0 = ⟦Assignment⟧(s,row);
  }
  equation ⟦MultipleAssignments[Assignment,Assignments]⟧(s,row) = row0
  {
    row1 = ⟦Assignment⟧(s,row);
    row0 = ⟦Assignments⟧(s,row1);
  }
}
function ⟦Assignment⟧: #State# × #Row# → #Row#
{
  equation ⟦TheAssignment[ID,Exp,Annotation]⟧(s,row) = row0
  {
    # int pos = Annotation.getPosition(); #
    value = ⟦Exp⟧(s);
    row0 = #row.set(pos, value)#;
  }
}
function ⟦DeleteStat⟧: #State# → #State#
{
  equation ⟦Delete[ID,WhereClause,Annotation]⟧(s) = s0
  {
    #
      TableSymbol tsym = Annotation.getTableSymbol();
      String name = tsym.id().name();
      Database.Content content = s.db().getContent(name);
      Table table = DatabaseUtils.table(content);
    #
      s1: #State# = #s.newTable(table)#;
      pred = Pred⟦WhereClause⟧(s1);
    #
      List<Row> rows = new ArrayList<Row>();
      for (Row row : table.list())
      {
        if (pred.test(row)) continue;
        rows.add(row);
      }
      Table table0 = new Table(rows);
      s.db().setContent(name, DatabaseUtils.content(table0));
    #
    s0 = s; // single state is passed through statements
  }
}
function ⟦SelectStat⟧: #State# → #Table#
{
  equation ⟦Select[TableExp,Annotation]⟧(s) = table
  {
    # ColumnSymbols csyms = Annotation.getColumnSymbols(); #
    table = ⟦TableExp⟧(s);
  }
  after
  {#
    if (s.out() != null) table.println(s.out(), csyms);
  #}
}
```

```
function ⟦TableExp⟧: #State# → #Table#
{
  equation ⟦TableExpValuesExp[ValuesExp]⟧(s) = table
  {
    table = ⟦ValuesExp⟧(s);
  }
  equation ⟦TableExpSelectExp[SelectExp]⟧(s) = table
  {
    table = ⟦SelectExp⟧(s);
  }
  equation ⟦TableExpCombination[TableExp1,SetOp,DistinctClause,TableExp2]⟧(s)
    = table
  {
    table1 = ⟦TableExp1⟧(s);
    table2 = ⟦TableExp2⟧(s);
    table3 = ⟦SetOp⟧(table1,table2);
    distinctRows: #Function<Table,Table># = #(Table t)->t.distinctRows()#;
    table = ⟦DistinctClause⟧(distinctRows,table3);
  }
}
function ⟦SetOp⟧: #Table# × #Table# → #Table#
{
  equation ⟦SetUnion⟧(table1,table2) = table
  {
    table = #table1.append(table2)#;
  }
  equation ⟦SetIntersect⟧(table1,table2) = table
  {
    #
      List<Row> list = new ArrayList<Row>();
      for (Row row : table1.list())
      {
        if (!table2.contains(row)) continue;
        list.add(row);
      }
    #
    table = #new Table(list)#;
  }
  equation ⟦SetExcept⟧(table1,table2) = table
  {
    #
      List<Row> list = new ArrayList<Row>();
      for (Row row : table1.list())
      {
        if (table2.contains(row)) continue;
        list.add(row);
      }
    #
    table = #new Table(list)#;
  }
}
function ⟦DistinctClause⟧: #Function<Table,Table># × #Table# → #Table#
{
  equation ⟦DistinctNone⟧(fun,table) = table0
```

```
  {
    table0 = #fun.apply(table)#;
  }
  equation ⟦DistinctAll⟧(fun,table) = table0
  {
    table0 = #table.allRows()#;
  }
  equation ⟦DistinctDistinct⟧(fun,table) = table0
  {
    table0 = #table.distinctRows()#;
  }
}
function ⟦ValuesExp⟧: #State# → #Table#
{
  equation ⟦TheValuesExp[RowExps]⟧(s) = table
  {
    table = ⟦RowExps⟧(s);
  }
}
function ⟦RowExps⟧: #State# → #Table#
{
  equation ⟦SingleRowExp[RowExp]⟧(s) = table
  {
    row = ⟦RowExp⟧(s);
    # List<Row> list = new ArrayList<Row>(); #
    # list.add(row); #
    table = #new Table(list)#;
  }
  equation ⟦MultipleRowExps[RowExp,RowExps]⟧(s) = table
  {
    row = ⟦RowExp⟧(s);
    rows = ⟦RowExps⟧(s);
    # List<Row> list = new ArrayList<Row>(); #
    # list.add(row); list.addAll(rows.list()); #
    table = #new Table(list)#;
  }
}
function ⟦RowExp⟧: #State# → #Row#
{
  equation ⟦TheRowExp[Exps]⟧(s) = row
  {
    vs = ⟦Exps⟧(s);
    row = #new Row(vs.list())#;
  }
}
function ⟦SelectExp⟧: #State# → #Table#
{
  equation ⟦TheSelectExp[DistinctClause,SelectClause,FromClause,
    WhereClause,GroupByClause,HavingClause,OrderByClause]⟧(s) = table
  {
    table1 = ⟦FromClause⟧(s);
    s0: #State# = #s.newTable(table1)#;
    table2 = ⟦WhereClause⟧(s0,table1);
    tables3 = ⟦GroupByClause⟧(table2);
```

```
    tables4 = ⟦HavingClause⟧(s0,tables3);
    table5 = ⟦SelectClause⟧(s0,tables4);
    allRows: #Function<Table,Table># = #(Table t)->t.allRows()#;
    table6 = ⟦DistinctClause⟧(allRows,table5);
    table = ⟦OrderByClause⟧(table6);
  }
}
function ⟦FromClause⟧: #State# → #Table#
{
  equation ⟦NoFromClause⟧(s) = table
  {
    table = #new Table()#;
  }
  equation ⟦SomeFromClause[TableD]⟧(s) = table
  {
    table = ⟦TableD⟧(s);
  }
}
function ⟦TableD⟧: #State# -> #Table#
{
  equation ⟦TableId[ID,AsClause,Annotation]⟧(s) = table
  {
    #
      TableSymbol tsym = Annotation.getTableSymbol();
      String name = tsym.id().name();
      Database.Content content = s.db().getContent(name);
    #
    table = # DatabaseUtils.table(content)#;
  }
  equation ⟦TableTableExp[TableExp,AsClause]⟧(s) = table
  {
    table = ⟦TableExp⟧(s);
  }
  equation ⟦TableJoin[TableD1,JoinClause,TableD2,JoinCondition,Annotation]⟧(s) = table
  {
    # ColumnSymbols c1 = Annotation.getColumnSymbols1(); #
    # ColumnSymbols c2 = Annotation.getColumnSymbols2(); #
    table1 = ⟦TableD1⟧(s);
    table2 = ⟦TableD2⟧(s);
    n1: #Integer# = #c1.list().size()#;
    n2: #Integer# = #c2.list().size()#;
    JoinCondition0 = ⟦JoinCondition⟧(s);
    table = ⟦JoinClause⟧(table1,table2,n1,n2,JoinCondition0);
  }
  equation ⟦TableCrossJoin[TableD1,CrossJoinClause,TableD2]⟧(s) = table
  {
    table1 = ⟦TableD1⟧(s);
    table2 = ⟦TableD2⟧(s);
    table = #table1.crossJoin(table2)#;
  }
}
function ⟦JoinClause⟧: #Table# × #Table# × #Integer# × #Integer# ×
  #Predicate<Row># → #Table#
{
```

```
  equation ⟦Join⟧(table1,table2,n1,n2,pred) = table
  {
    table = #crossJoin(table1, table2, pred)#;
  }
  equation ⟦InnerJoin⟧(table1,table2,n1,n2,pred) = table
  {
    table = #crossJoin(table1, table2, pred)#;
  }
  equation ⟦LeftJoin⟧(table1,table2,n1,n2,pred) = table
  {
    table = #leftJoin(table1, table2, n2, pred)#;
  }
  equation ⟦LeftOuterJoin⟧(table1,table2,n1,n2,pred) = table
  {
    table = #leftJoin(table1, table2, n2, pred)#;
  }
  equation ⟦RightJoin⟧(table1,table2,n1,n2,pred) = table
  {
    table = #rightJoin(table1, n1, table2, pred)#;
  }
  equation ⟦RightOuterJoin⟧(table1,table2,n1,n2,pred) = table
  {
    table = #rightJoin(table1, n1, table2, pred)#;
  }
  equation ⟦FullJoin⟧(table1,table2,n1,n2,pred) = table
  {
    table = #fullJoin(table1, n1, table2, n2, pred)#;
  }
  equation ⟦FullOuterJoin⟧(table1,table2,n1,n2,pred) = table
  {
    table = #fullJoin(table1, n1, table2, n2, pred)#;
  }
}
function ⟦JoinCondition⟧: #State# → #Predicate<Row>#
{
  equation⟦JoinOn[Exp]⟧(s) = pred
  {
    # Predicate<Row> pred0 = (Row row)->
    {
    #
      s0: #State# = #s.newRow(row)#;
      v = ⟦Exp⟧(s0);
    #
      return v.isTrue();
    };
    #
    pred = #pred0#;
  }
  equation⟦JoinUsing[IDs,Annotation]⟧(s) = pred
  {
    # List<Integer> pos1 = Annotation.getPositionSeq1(); #
    # List<Integer> pos2 = Annotation.getPositionSeq2(); #
    # int n = pos1.size(); #
    pred = #(Row row)->
```

```
    {
      for (int i = 0; i < n; i++)
      {
        Value value1 = row.list().get(pos1.get(i));
        Value value2 = row.list().get(pos2.get(i));
        if (value1 instanceof Value.Null) return false;
        if (value2 instanceof Value.Null) return false;
        if (!value1.equals(value2)) return false;
      }
      return true;
    }#;
  }
}
function ⟦WhereClause⟧: #State# × #Table# → #Table#
{
  equation ⟦NoWhere⟧(s,table) = table0
  {
    table0 = table;
  }
  equation ⟦SomeWhere[Exp]⟧(s,table) = table0
  {
    #
      Predicate<Row> pred = (Row row)->
      {
    #
        s0: #State# = #s.newRow(row)#;
        v = ⟦Exp⟧(s0);
    #
        return v.isTrue();
      };
    #
    table0 = #table.filter(pred)#;
  }
}
function Pred⟦WhereClause⟧: #State# → #Predicate<Row>#
{
  equation Pred⟦NoWhere⟧(s) = pred
  {
    pred = #(Row row)->true#;
  }
  equation Pred⟦SomeWhere[Exp]⟧(s) = pred
  {
    #
      Predicate<Row> pred0 = (Row row)->
      {
    #
        s0: #State# = #s.newRow(row)#;
        v = ⟦Exp⟧(s0);
    #
        return v.isTrue();
      };
    #
    pred = #pred0#;
  }
```

```
}
function ⟦GroupByClause⟧: #Table# → #Tables#
{
  equation ⟦NoGroupBy⟧(table) = tables
  {
    # List<Table> list = new ArrayList<Table>(); #
    # list.add(table); #
    tables = #new Tables(list)#;
  }
  equation ⟦SomeGroupBy[ColumnRefs,Annotation]⟧(table) = tables
  {
    #
      List<Table> list = new ArrayList<Table>();
      List<Integer> pos = Annotation.getPositionSeq();
      Map<Row,List<Row>> map = new LinkedHashMap<Row,List<Row>>();
      for (Row row : table.list())
      {
        Row key = row.project(pos);
        List<Row> rows = map.get(key);
        if (rows == null) { rows = new ArrayList<Row>(); map.put(key, rows); }
        rows.add(row);
      }
      for (List<Row> rows : map.values()) list.add(new Table(rows));
    #
    tables = #new Tables(list)#;
  }
}
function ⟦HavingClause⟧: #State# × #Tables# → #Tables#
{
  equation ⟦NoHaving⟧(s,tables) = tables0
  {
    tables0 = tables;
  }
  equation ⟦SomeHaving[Exp]⟧(s,tables) = tables0
  {
    #
      Predicate<Table> pred = (Table table)->
      {
    #
        s0: #State# = #s.newRow(table.list().get(0))#;
        v = ⟦Exp⟧(s0);
    #
        return v.isTrue();
      };
    #
    tables0 = #tables.filter(pred)#;
  }
}
function ⟦SelectClause⟧: #State# × #Tables# → #Table#
{
  equation ⟦TheSelectClause[ColumnExps,Annotation]⟧(s,tables) = table
  {
    #
      Boolean grouped = (Boolean)Annotation.getValue();
```

```
      List<Row> list = new ArrayList<Row>();
      for (Table table0 : tables.list())
      {
    #
        s0: #State# = #s.newTable(table0)#;
        table1 = ⟦ColumnExps⟧(s0);
    #
        if (grouped && !table1.list().isEmpty())
          list.add(table1.list().get(0));
        else
          list.addAll(table1.list());
      }
    #
    table = #new Table(list)#;
  }
}
function ⟦ColumnExps⟧: #State# → #Table#
{
  equation ⟦SingleColumnExp[ColumnExp]⟧(s) = table
  {
    table = ⟦ColumnExp⟧(s);
  }
  equation ⟦MultipleColumnExps[ColumnExp,ColumnExps]⟧(s) = table
  {
     table1 = ⟦ColumnExp⟧(s);
     table2 = ⟦ColumnExps⟧(s);
     #
       List<Row> list = new ArrayList<Row>();
       int n = s.currentTable().list().size();
       for (int i = 0; i < n; i++)
       {
         Row row1 = table1.list().get(i);
         Row row2 = table2.list().get(i);
         list.add(row1.append(row2));
       }
     #
     table = #new Table(list)#;
  }
}
function ⟦ColumnExp⟧: #State# → #Table#
{
  equation ⟦ColumnExpAll⟧(s) = table
  {
    table = #s.currentTable()#;
  }
  equation ⟦ColumnExpId[ID,Annotation]⟧(s) = table
  {
    #
      List<Integer> pos = Annotation.getPositionSeq();
      Table table0 = s.currentTable();
      List<Row> list = new ArrayList<Row>();
      for (Row row : table0.list()) list.add(row.project(pos));
    #
    table = #new Table(list)#;
```

```
  }
  equation ⟦ColumnExpExp[Exp,AsClause]⟧(s) = table
  {
    #
      Table table0 = s.currentTable();
      List<Row> list = new ArrayList<Row>();
      for (Row row : table0.list())
      {
    #
        s0: #State# = #s.newRow(row)#;
        v = ⟦Exp⟧(s0);
    #
        List<Value> values;
        switch (v)
        {
          case Value.Seq(var vs) -> { values = vs.list(); }
          default -> { values = new ArrayList<Value>(); values.add(v); }
        }
        list.add(new Row(values));
      }
    #
    table = #new Table(list)#;
  }
}
function ⟦OrderByClause⟧: #Table# → #Table#
{
  equation ⟦OrderByNone⟧(table) = table0
  {
    table0 = table;
  }
  equation ⟦OrderBySome[OrderByColumns,Annotation]⟧(table) = table0
  {
    #
      List<Integer> pos = Annotation.getPosSeq();
      List<Boolean> asc = Annotation.getAscSeq();
    #
    table0 = #table.sort(pos, asc)#;
  }
}
function ⟦Exps⟧: #State# → #Values#
{
  equation ⟦SingleExp[Exp]⟧(s) = vs
  {
    v = ⟦Exp⟧(s);
    # List<Value> list = new ArrayList<Value>(); #
    # list.add(v); #
    vs = #new Values(list)#;
  }
  equation ⟦MultipleExps[Exp,Exps]⟧(s) = vs
  {
    v = ⟦Exp⟧(s);
    vs0 = ⟦Exps⟧(s);
    # List<Value> list = new ArrayList<Value>(); #
    # list.add(v); list.addAll(vs0.list()); #
```

```
    vs = #new Values(list)#;
  }
}
function ⟦Exp⟧: #State# → #Value#
{
  equation ⟦ExpMetaVar[ID]⟧(s) = v
  {
    v = #s.store().map().get(new Id(ID))#;
  }
  equation ⟦ExpLiteral[Literal,Annotation]⟧(s) = v
  {
    v = #(Value)Annotation.getValue()#;
  }
  equation ⟦ExpRef[ColumnRef,Annotation]⟧(s) = v
  {
    #
      Integer pos = Annotation.getColumnPos();
      Kind kind = Annotation.getColumnKind();
      Integer depth = Annotation.getColumnDepth();
    #
    v = #switch (kind) {
           case Kind.cell -> s.rows().list().get(depth).list().get(pos);
           case Kind.column -> {
            List<Value> values = new ArrayList<Value>();
            for (Row row : s.currentTable().list())
              values.add(row.list().get(pos));
            yield new Value.Seq(new Values(values));
            }
          }#;
  }
  equation ⟦ExpUnary[UnaryOp,Exp]⟧(s) = v
  {
    E: #Function<State,Value># = #null#;
    #E = (State s0)->{# s1: #State# = #s0#; v0 = ⟦Exp⟧(s1); #return v0;};#
    v = ⟦UnaryOp⟧(s,E);
  }
  equation ⟦ExpBinary[Exp1,BinaryOp,Exp2]⟧(s) = v
  {
    E1: #Function<State,Value># = #null#;
    #E1 = (State s0)->{# s1: #State# = #s0#; v1 = ⟦Exp1⟧(s1); #return v1;};#
    E2: #Function<State,Value># = #null#;
    #E2 = (State s0)->{# s2: #State# = #s0#; v2 = ⟦Exp2⟧(s2); #return v2;};#
    v = ⟦BinaryOp⟧(s,E1,E2);
  }
  equation ⟦ExpExps[Exps]⟧(s) = v
  {
    vs = ⟦Exps⟧(s);
    # List<Value> list = vs.list(); #
    v = #list.size() == 1 ? list.get(0) : new Value.Seq(vs)#;
  }
  equation ⟦ExpSelect[SelectExp]⟧(s) = v
  {
    table = ⟦SelectExp⟧(s);
    # subsql.Semantics.check(table.list().size() == 1,
```

```
      "selected table has not 1 row, but " + table.list().size()); #
  # List<Value> list = table.list().get(0).list(); #
  v = #list.size() == 1 ? list.get(0) :
    new Value.Seq(new Values(table.list().get(0).list()))#;
}
equation ⟦ExpBetween[Exp1,NotClause,Exp2,Exp3]⟧(s) = v
{
  v1 = ⟦Exp1⟧(s);
  E2: #Function<State,Value># = #null#;
  #E2 = (State s0)->{#
    v2 = ⟦Exp2⟧(s);
    # return new Value.Bool(v2.lessEqual(v1)); };#
  E3: #Function<State,Value># = #null#;
  #E3 = (State s0)->{#
    v3 = ⟦Exp3⟧(s);
    # return new Value.Bool(v1.lessEqual(v3)); };#
  v = #binaryAnd(s,E2,E3)#;
}
equation ⟦ExpIsNull[Exp,NotClause]⟧(s) = v
{
  v1 = ⟦Exp⟧(s);
  b: #Boolean# = #v1 instanceof Value.Null#;
  v0: #Value# = #new Value.Bool(b)#;
  v = ⟦NotClause⟧(v0);
}
equation ⟦ExpIsTrue[Exp,NotClause]⟧(s) = v
{
  v1 = ⟦Exp⟧(s);
  b: #Boolean# = #v1 instanceof Value.Bool && ((Value.Bool)v1).bool()#;
  v0: #Value# = #v1 instanceof Value.Null ? v1 : new Value.Bool(b)#;
  v = ⟦NotClause⟧(v0);
}
equation ⟦ExpIsFalse[Exp,NotClause]⟧(s) = v
{
  v1 = ⟦Exp⟧(s);
  b: #Boolean# = #v1 instanceof Value.Bool && !((Value.Bool)v1).bool()#;
  v0: #Value# = #v1 instanceof Value.Null ? v1 : new Value.Bool(b)#;
  v = ⟦NotClause⟧(v0);
}
equation ⟦ExpIsDistinct[Exp1,NotClause,Exp2]⟧(s) = v
{
  v1 = ⟦Exp1⟧(s);
  v2 = ⟦Exp1⟧(s);
  b: #Boolean# = #!v1.equals(v2)#;
  v0: #Value# = #new Value.Bool(b)#;
  v = ⟦NotClause⟧(v0);
}
equation ⟦ExpLike[Exp,STR,Annotation]⟧(s) = v
{
  # java.util.regex.Pattern p = Annotation.getPattern(); #
  v1 = ⟦Exp⟧(s);
  b: #Boolean# = #v1 instanceof Value.String &&
    p.matcher(((Value.String)v1).string()).matches()#;
  v = #v1 instanceof Value.Null ? v1 : new Value.Bool(b)#;
```

```
}
equation ⟦ExpCase[WhenClauses,ElseClause]⟧(s) = v
{
  vopt = ⟦WhenClauses⟧(s);
  # if (vopt.optional().isPresent()) #
    v = #vopt.optional().get()#;
  # else { #
    v0 = ⟦ElseClause⟧(s);
  # v = v0; #
  # } #
}
equation ⟦ExpCaseExp[Exp,WhenClauses,ElseClause]⟧(s) = v
{
  v0 = ⟦Exp⟧(s);
  vopt = Exp⟦WhenClauses⟧(s,v0);
  # if (vopt.optional().isPresent()) #
    v = #vopt.optional().get()#;
  # else { #
    v0 = ⟦ElseClause⟧(s);
  # v = v0; #
  # } #
}
equation ⟦ExpNullIf[Exp1,Exp2]⟧(s) = v
{
  v1 = ⟦Exp1⟧(s);
  v2 = ⟦Exp2⟧(s);
  v = #v1.equals(v2) ? new Value.Null() : v1#;
}
equation ⟦ExpCoalesce[Exps]⟧(s) = v
{
  vs = ⟦Exps⟧(s);
  #
    for (Value v0 : vs.list())
    {
      if (v0 instanceof Value.Null) continue;
      return v0;
    }
  #
  v = #new Value.Null()#;
}
equation ⟦ExpExists[SelectExp]⟧(s) = v
{
  table = ⟦SelectExp⟧(s);
  v = #new Value.Bool(table.list().size() > 0)# ;
}
equation ⟦ExpIn[Exp,NotClause,Exps]⟧(s) = v
{
  v0 = ⟦Exp⟧(s);
  vs = ⟦Exps⟧(s);
  v1: #Value# = #v0.isIn(vs)#;
  v = ⟦NotClause⟧(v1);
}
equation ⟦ExpInSelect[Exp,NotClause,SelectExp]⟧(s) = v
{
```

```
    v0 = ⟦Exp⟧(s);
    table = ⟦SelectExp⟧(s);
    #
      List<Value> list = new ArrayList<Value>();
      for (Row row : table.list()) list.add(row.list().get(0));
      Values vs = new Values(list);
    #
    v = #v0.isIn(vs)#;
  }
  equation ⟦ExpQuantified[Exp,BinaryOp,Quantifier,SelectExp]⟧(s) = v
  {
    v0 = ⟦Exp⟧(s);
    table = ⟦SelectExp⟧(s);
    # List<Value> list = new ArrayList<Value>(); #
    # for (Row row : table.list()) list.add(row.list().get(0)); #
    vs: #Values# = #new Values(list)#;
    ea: #Function<State,Value># = #(State s0)->v0#;
    pred: #Function<Value,Value># = #null#;
    # pred = (Value v1)->{ #
        eb: #Function<State,Value># = #(State s0)->v1#;
        vc = ⟦BinaryOp⟧(s,ea,eb);
        # return vc; }; #
    v = ⟦Quantifier⟧(vs,pred);
  }
  equation ⟦ExpAggregateExp[AggregateExp]⟧(s) = v
  {
    v = ⟦AggregateExp⟧(s);
  }
}
function ⟦UnaryOp⟧: #State# × #Function<State,Value># → #Value#
{
  equation ⟦UnaryNot⟧(s,e) = v
  {
    v = #switch (e.apply(s)) {
           case Value.Bool(var b) -> new Value.Bool(!b);
           default -> new Value.Null();
         }#;
  }
  equation ⟦UnaryMinus⟧(s,e) = v
  {
    v = #switch (e.apply(s)) {
           case Value.Int(var i) -> new Value.Int(-i);
           default -> new Value.Null();
         }#;
  }
}
function ⟦BinaryOp⟧: #State# ×
  #Function<State,Value># × #Function<State,Value># → #Value#
{
  equation ⟦BinaryPlus⟧(s,e1,e2) = v
  { v = #intFunction(s, e1, e2, (Integer i1, Integer i2)->i1+i2)#; }
  equation ⟦BinaryMinus⟧(s,e1,e2) = v
  { v = #intFunction(s, e1, e2, (Integer i1, Integer i2)->i1-i2)#; }
  equation ⟦BinaryTimes⟧(s,e1,e2) = v
```

```
  { v = #intFunction(s, e1, e2, (Integer i1, Integer i2)->i1*i2)#; }
  equation ⟦BinaryDiv⟧(s,e1,e2) = v
  { v = #intFunction(s, e1, e2, (Integer i1, Integer i2)->i1/i2)#; }
  equation ⟦BinaryMod⟧(s,e1,e2) = v
  { v = #intFunction(s, e1, e2, (Integer i1, Integer i2)->i1%i2)#; }
  equation ⟦BinaryBitAnd⟧(s,e1,e2) = v
  { v = #intFunction(s, e1, e2, (Integer i1, Integer i2)->i1&i2)#; }
  equation ⟦BinaryBitOr⟧(s,e1,e2) = v
  { v = #intFunction(s, e1, e2, (Integer i1, Integer i2)->i1|i2)#; }
  equation ⟦BinaryBitExclOr⟧(s,e1,e2) = v
  { v = #intFunction(s, e1, e2, (Integer i1, Integer i2)->i1^i2)#; }
  equation ⟦BinaryEqual⟧(s,e1,e2) = v
  { v = #valuePred(s, e1, e2, (Value v1, Value v2)->v1.equals(v2))#; }
  equation ⟦BinaryGreater⟧(s,e1,e2) = v
  { v = #valuePred(s, e1, e2, (Value v1, Value v2)->v2.less(v1))#; }
  equation ⟦BinaryGreaterEqual⟧(s,e1,e2) = v
  { v = #valuePred(s, e1, e2, (Value v1, Value v2)->v2.lessEqual(v1))#; }
  equation ⟦BinaryLess⟧(s,e1,e2) = v
  { v = #valuePred(s, e1, e2, (Value v1, Value v2)->v1.less(v2))#; }
  equation ⟦BinaryLessEqual⟧(s,e1,e2) = v
  { v = #valuePred(s, e1, e2, (Value v1, Value v2)->v1.lessEqual(v2))#; }
  equation ⟦BinaryNotEqual⟧(s,e1,e2) = v
  { v = #valuePred(s, e1, e2, (Value v1, Value v2)->!v1.equals(v2))#; }
  equation ⟦BinaryAnd⟧(s,e1,e2) = v
  { v = #binaryAnd(s, e1, e2)#; }
  equation ⟦BinaryOr⟧(s,e1,e2) = v
  { v = #binaryOr(s, e1, e2)#; }
}
function ⟦NotClause⟧: #Value# → #Value#
{
  equation ⟦NoNot⟧(v) = v0
  {
    v0 = v;
  }
  equation ⟦SomeNot⟧(v) = v0
  {
    v0 = #switch (v)
    {
      case Value.Bool(var b) -> new Value.Bool(!b);
      default -> new Value.Null();
    }#;
  }
}
function ⟦Quantifier⟧: #Values# × #Function<Value,Value># → #Value#
{
  equation ⟦QuantifierAny⟧(vs,p) = v
  {
    #
      Value nullValue = null;
      for (Value v0 : vs.list())
      {
        Value v1 = p.apply(v0);
        if (v1 instanceof Value.Null) nullValue = v1;
        if (!(v1 instanceof Value.Bool)) continue;
```

```
        if (((Value.Bool)v1).bool()) return v1;
      }
      if (nullValue != null) return nullValue;
    #
    v = #new Value.Bool(false)#;
  }
  equation ⟦QuantifierSome⟧(vs,p) = v
  {
     #
      Value nullValue = null;
      for (Value v0 : vs.list())
      {
        Value v1 = p.apply(v0);
        if (v1 instanceof Value.Null) nullValue = v1;
        if (!(v1 instanceof Value.Bool)) continue;
        if (((Value.Bool)v1).bool()) return v1;
      }
      if (nullValue != null) return nullValue;
    #
    v = #new Value.Bool(false)#;
  }
  equation ⟦QuantifierAll⟧(vs,p) = v
  {
    #
      Value nullValue = null;
      for (Value v0 : vs.list())
      {
        Value v1 = p.apply(v0);
        if (v1 instanceof Value.Null) nullValue = v1;
        if (!(v1 instanceof Value.Bool)) continue;
        if (!((Value.Bool)v1).bool()) return v1;
      }
      if (nullValue != null) return nullValue;
    #
    v = #new Value.Bool(true)#;
  }
}
function ⟦WhenClauses⟧: #State# → #ValueOption#
{
   equation ⟦SingleWhen[WhenClause]⟧(s) = vopt
   {
     vopt = ⟦WhenClause⟧(s);
   }
   equation ⟦MultipleWhens[WhenClause,WhenClauses]⟧(s) = vopt
   {
     vopt0 = ⟦WhenClause⟧(s);
     # if (vopt0.optional().isPresent()) #
         vopt = #vopt0#;
     # else { #
         vopt1 = ⟦WhenClauses⟧(s);
     #   vopt = vopt1; #
     # } #
   }
}
```

```
function Exp⟦WhenClauses⟧: #State# × #Value# → #ValueOption#
{
   equation Exp⟦SingleWhen[WhenClause]⟧(s,v) = vopt
   {
     vopt = Exp⟦WhenClause⟧(s,v);
   }
   equation Exp⟦MultipleWhens[WhenClause,WhenClauses]⟧(s,v) = vopt
   {
     vopt0 = Exp⟦WhenClause⟧(s,v);
     # if (vopt0.optional().isPresent()) #
         vopt = #vopt0#;
     # else { #
         vopt1 = Exp⟦WhenClauses⟧(s,v);
     #   vopt = vopt1; #
     # } #
   }
}
function ⟦WhenClause⟧: #State# → #ValueOption#
{
  equation ⟦TheWhenClause[Exp1,Exp2]⟧(s) = vopt
  {
    v1 = ⟦Exp1⟧(s);
    #
      switch (v1) {
        case Value.Bool(var b) -> {
          if (b) {
    #
            v2 = ⟦Exp2⟧(s);
            vopt = #new ValueOption(Optional.of(v2))#;
    #
         }
         else { vopt = new ValueOption(Optional.empty()); }
        }
        default ->
        { vopt = new ValueOption(Optional.of(new Value.Null())); }
      }
    #
  }
}
function Exp⟦WhenClause⟧: #State# × #Value# → #ValueOption#
{
  equation Exp⟦TheWhenClause[Exp1,Exp2]⟧(s,v) = vopt
  {
    # if (v instanceof Value.Null) #
      vopt = #new ValueOption(Optional.of(new Value.Null()))#;
    # else { #
        v1 = ⟦Exp1⟧(s);
    #   if (v1 instanceof Value.Null)
          vopt = new ValueOption(Optional.of(new Value.Null()));
        else if (v1.equals(v)) { #
         v2 = ⟦Exp2⟧(s);
    #
         vopt = new ValueOption(Optional.of(v2));
        } else vopt = new ValueOption(Optional.empty());
```

```
      }
    #
  }
}
function ⟦ElseClause⟧: #State# → #Value#
{
  equation ⟦NoElse⟧(s) = v
  {
    v = #new Value.Null()#;
  }
  equation ⟦SomeElse[Exp]⟧(s) = v
  {
    v = ⟦Exp⟧(s);
  }
}
function ⟦AggregateExp⟧: #State# → #Value#
{
  equation ⟦AggregateExpFun[AggregateFun,AggregateArg]⟧(s) = v
  {
    vs = ⟦AggregateArg⟧(s);
    v = ⟦AggregateFun⟧(vs);
  }
  equation ⟦AggregateExpCount⟧(s) = v
  {
    v = #new Value.Int(s.currentTable().list().size())#;
  }
}
function ⟦AggregateArg⟧: #State# → #Values#
{
  equation ⟦TheAggregateArg[DistinctClause,Exp]⟧(s) = vs
  {
    v = ⟦Exp⟧(s);
    vs1: #Values# = #null#;
    #
      switch(v) {
        case Value.Seq(var vs0) -> { vs1 = vs0; }
        default -> {
          List<Value> list = new ArrayList<Value>();
          list.add(v);
          vs1 = new Values(list);
        }
      }
      List<Row> rows = new ArrayList<Row>();
      for (Value value : vs1.list())
      {
        List<Value> values = new ArrayList<Value>();
        values.add(value);
        rows.add(new Row(values));
      }
    #
    table1: #Table# = #new Table(rows)#;
    allRows: #Function<Table,Table># = #(Table t)->t.allRows()#;
    table2 = ⟦DistinctClause⟧(allRows,table1);
    #
```

```
      List<Value> values = new ArrayList<Value>();
      for (Row row : table2.list()) values.add(row.list().get(0));
    #
    vs = #new Values(values)#;
  }
}
function ⟦AggregateFun⟧: #Values# → #Value#
{
  equation ⟦AggregateMin⟧(vs) = v
  {
    # subsql.Semantics.check(vs.list().size() > 0,
        "MIN is applied to empty list of values"); #
    v = #vs.list().get(0)#;
    #
      for (Value v0 : vs.list())
      {
        if (v0.less(v)) v = v0;
      }
    #
  }
  equation ⟦AggregateMax⟧(vs) = v
  {
    # subsql.Semantics.check(vs.list().size() > 0,
        "MIN is applied to empty list of values"); #
    v = #vs.list().get(0)#;
    #
      for (Value v0 : vs.list())
      {
        if (v.less(v0)) v = v0;
      }
    #
  }
  equation ⟦AggregateSum⟧(vs) = v
  {
    #
      int sum = 0;
      for (Value v0 : vs.list())
      {
        if (!(v0 instanceof Value.Int)) continue;
        sum += ((Value.Int)v0).integer();
      }
    #
    v = #new Value.Int(sum)#;
  }
  equation ⟦AggregateAvg⟧(vs) = v
  {
    #
      int count = vs.list().size();
      subsql.Semantics.check(count > 0,
        "AVG is applied to empty list of values");
      int sum = 0;
      for (Value v0 : vs.list())
      {
        if (!(v0 instanceof Value.Int)) continue;
```

```
          sum += ((Value.Int)v0).integer();
        }
      #
      v = #new Value.Int(sum/count)#;
    }
    equation ⟦AggregateCount⟧(vs) = v
    {
      v = #new Value.Int(vs.list().size())#;
    }
  }
}
// ------------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------------>0
```

B.2 Type System Utilities

```
// ------------------------------------------------------------------------
// TypeSystem.txt
// Auxiliaries for the SubSQL type checker.
// (c) 2024 https://www.risc.jku.at/research/formal/software/SLANG
// Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <william.steingartner@tuke.sk>
// ------------------------------------------------------------------------
package subsql;

import java.util.*;
import java.util.function.*;
import java.util.regex.*;

import static subsql.Semantics.Value;

public class TypeSystem
{
  // the type checker environment
  public static record Env(
    TableEnv db,                      // all the tables in the database
    VarEnv venv,                      // the variables in the store
    TableEnv tenv,                    // the currently visible tables
    List<ColumnSymbols> cstack,       // the columns stack, top is "current" table
    ColumnEnv cenv,                   // the visible columns
    Map<ColumnSymbol,Kind> ckind,     // column looked up in row or in table
    Map<ColumnSymbol,Integer> cdepth, // the stack position for the lookup
    int depth,                        // the current stack depth
    Set<ColumnSymbol> gsyms           // the grouped columns
  )
  {
    public ColumnSymbols csyms() { return cstack.getLast(); }
    public Env()
    {
      this(new TableEnv(), new VarEnv(), new TableEnv(),
          new ArrayList<ColumnSymbols>(), new ColumnEnv(),
          new HashMap<ColumnSymbol,Kind>(),
```

```
      new HashMap<ColumnSymbol,Integer>(), 0,
      new HashSet<ColumnSymbol>());

}
public static Env setDBEnv(Env env)
{
  // set database environment to table view
  return new Env(env.tenv, env.venv, env.tenv,
      env.cstack, env.cenv, env.ckind, env.cdepth, env.depth, env.gsyms);
}
public static Env clearTableEnv(Env env)
{
  // set table view to empty
  return new Env(env.db, env.venv, new TableEnv(),
      env.cstack, env.cenv, env.ckind, env.cdepth, env.depth, env.gsyms);
}
public static Env setTableEnv(Env env)
{
  // set table environment to database view
  return new Env(env.db, env.venv, env.db,
      env.cstack, env.cenv, env.ckind, env.cdepth, env.depth, env.gsyms);
}

public static Env emptyEnv(Env env)
{
  // empty environment with database view preserved
  return new Env(env.db(),
      new VarEnv(), new TableEnv(),
      new ArrayList<ColumnSymbols>(),
      new ColumnEnv(),
      new HashMap<ColumnSymbol,Kind>(),
      new HashMap<ColumnSymbol,Integer>(), 0,
      new HashSet<ColumnSymbol>());
}
public Env put(Id key, VarSymbol value)
{
  VarEnv venv0 = venv.put(key, value);
  return new Env(db, venv0, tenv, cstack, cenv, ckind, cdepth, depth, gsyms);
}
public Env put(Id key, TableSymbol value)
{
  TableEnv tenv0 = tenv.put(key, value);
  return new Env(db, venv, tenv0, cstack, cenv, ckind, cdepth, depth, gsyms);
}
public Env addTable(Id id, TableSymbol tsym)
{
  return put(id, tsym);
}
public Env removeTable(Id key)
{
  TableEnv tenv0 = tenv.remove(key);
  return new Env(db, venv, tenv0, cstack, cenv, ckind, cdepth, depth, gsyms);
}
public Env enterTable(ColumnSymbols csyms)
```

```
  {
    ColumnEnv cenv0 = cenv.addColumns(csyms);
    return enterTable(cenv0, csyms);
  }
  public Env enterTable(ColumnEnv cenv0, ColumnSymbols csyms0)
  {
    List<ColumnSymbols> cstack0 = new ArrayList<ColumnSymbols>(cstack);
    cstack0.add(csyms0);
    Map<ColumnSymbol,Kind> ckind0 = new HashMap<ColumnSymbol,Kind>(ckind);
    for (ColumnSymbol csym : csyms0.list()) ckind0.put(csym, Kind.column);
    Map<ColumnSymbol,Integer> cdepth0 = new HashMap<ColumnSymbol,Integer>(cdepth);
    for (ColumnSymbol csym : csyms0.list()) cdepth0.put(csym, depth);
    return new Env(db, venv, tenv, cstack0, cenv0, ckind0, cdepth0, depth+1, gsyms);
  }
  public Env enterTable()
  {
    Map<ColumnSymbol,Kind> ckind0 = new HashMap<ColumnSymbol,Kind>(ckind);
    for (ColumnSymbol csym : csyms().list()) ckind0.put(csym, Kind.column);
    return new Env(db, venv, tenv, cstack, cenv, ckind0, cdepth, depth, gsyms);
  }
  public Env enterRow()
  {
    Map<ColumnSymbol,Kind> ckind0 = new HashMap<ColumnSymbol,Kind>(ckind);
    for (ColumnSymbol csym : csyms().list()) ckind0.put(csym, Kind.cell);
    return new Env(db, venv, tenv, cstack, cenv, ckind0, cdepth, depth, gsyms);
  }
  public Env addColumn(Id id, ColumnSymbol csym)
  {
    Map<Id,ColumnSymbol> cenv0 = new HashMap<Id,ColumnSymbol>(cenv.map());
    cenv0.put(id, csym);
    return new Env(db, venv, tenv, cstack, new ColumnEnv(cenv0), ckind, cdepth,
        depth, gsyms);
  }
  public Env removeColumn(Id id)
  {
    Map<Id,ColumnSymbol> cenv0 = new HashMap<Id,ColumnSymbol>(cenv.map());
    cenv0.remove(id);
    return new Env(db, venv, tenv, cstack, new ColumnEnv(cenv0), ckind, cdepth,
        depth, gsyms);
  }
  public Env clearColumns()
  {
    Map<Id,ColumnSymbol> cenv0 = new HashMap<Id,ColumnSymbol>();
    return new Env(db, venv, tenv, cstack, new ColumnEnv(cenv0), ckind, cdepth,
        depth, gsyms);
  }
  public Env setGrouped(ColumnSymbols csyms0)
  {
    Set<ColumnSymbol> gsyms = new HashSet<ColumnSymbol>(csyms0.list());
    return new Env(db, venv, tenv, cstack, cenv, ckind, cdepth, depth, gsyms);
  }
}

public static record VarEnv(Map<Id,VarSymbol> map)
```

```
{
  public VarEnv() { this(new HashMap<Id,VarSymbol>()); }
  public VarEnv put(Id key, VarSymbol value)
  {
    Map<Id,VarSymbol> map0 = new HashMap<Id,VarSymbol>(map);
    map0.put(key, value);
    return new VarEnv(map0);
  }
}
public static record VarSymbol(Id id, VarType type) { }
public static enum VarType { intvar, boolvar, stringvar }

public static record TableEnv(Map<Id,TableSymbol> map)
{
  public TableEnv() { this(new HashMap<Id,TableSymbol>()); }
  public TableEnv put(Id key, TableSymbol value)
  {
    Map<Id,TableSymbol> map0 = new HashMap<Id,TableSymbol>(map);
    map0.put(key, value);
    return new TableEnv(map0);
  }
  public TableEnv remove(Id key)
  {
    Map<Id,TableSymbol> map0 = new HashMap<Id,TableSymbol>(map);
    map0.remove(key);
    return new TableEnv(map0);
  }
}
public static record TableSymbol(Id id, ColumnSymbols csyms) { }

public static record ColumnEnv(Map<Id,ColumnSymbol> map)
{
  public ColumnEnv() { this(new HashMap<Id,ColumnSymbol>()); }
  public ColumnEnv addColumns(ColumnSymbols csyms)
  {
    Map<Id,ColumnSymbol> map0 = new HashMap<Id,ColumnSymbol>(map);
    for (ColumnSymbol csym : csyms.list())
    {
      Id id = csym.id();
      if (map0.get(id) != null)
      {
        map0.remove(id);
        continue;
      }
      map0.put(id,csym);
    }
    return new ColumnEnv(map0);
  }
}
public static record ColumnSymbol(Id id, ColumnType ctype,
  ConstraintSet cons, IdOption topt)
{
  public boolean matchedBy(ColumnSymbol csym, boolean symmetric)
  {
```

```
    String name1 = id.name;
    String name2 = csym.id.name;
    if (!name1.equals("_") && !name2.equals("_") & !name1.equals(name2))
      return false;
    if (csym.ctype instanceof ColumnType.Null) return true;
    if (symmetric && ctype instanceof ColumnType.Null) return true;
    if (csym.ctype instanceof ColumnType.Int) return ctype instanceof ColumnType.Int;
    if (csym.ctype instanceof ColumnType.VarChar) return ctype instanceof ColumnType.VarChar;
    return false;
  }
  public boolean matchedBy(Type type0)
  {
    if (type0 instanceof Type.Null) return true;
    if (type0 instanceof Type.Int) return ctype instanceof ColumnType.Int;
    if (type0 instanceof Type.String) return ctype instanceof ColumnType.VarChar;
    return false;
  }
  public ColumnSymbol mergeWith(ColumnSymbol csym)
  {
    Id id0 = id.name.equals("_") ? csym.id : id;
    ColumnType type0 = (ctype instanceof ColumnType.Null) ? csym.ctype() : ctype;
    return new ColumnSymbol(id0,type0,new ConstraintSet(), new IdOption(Optional.empty()));
  }
}
public static record ColumnSymbols(List<ColumnSymbol> list)
{
  public ColumnSymbols() { this(new ArrayList<ColumnSymbol>()); }
  public ColumnSymbols add(ColumnSymbol... elems)
  {
    List<ColumnSymbol> list0 = new ArrayList<ColumnSymbol>(list);
    list0.addAll(Arrays.asList(elems));
    return new ColumnSymbols(list0);
  }
  public ColumnSymbols add(ColumnSymbols csyms)
  {
    List<ColumnSymbol> list0 = new ArrayList<ColumnSymbol>(list);
    list0.addAll(csyms.list);
    return new ColumnSymbols(list0);
  }
  public ColumnSymbols mergeWith(ColumnSymbols csyms0)
  {
    int n = list.size(); int n0 = csyms0.list().size();
    check(n == n0, "cannot combine table with " + n +
        " columns with table with " + n0 + " columns");
    List<ColumnSymbol> result = new ArrayList<ColumnSymbol>();
    for (int i = 0; i < n; i++)
    {
      ColumnSymbol csym1 = list.get(i);
      ColumnSymbol csym2 = csyms0.list().get(i);
      check(csym1.matchedBy(csym2, true),
          "cannot combine table whose column " + (i+1) +
          " has name " + csym1.id().name() + " and type " + csym1.ctype() +
          " with table whose column " + (i+1) +
          " has name " + csym2.id().name() + " and type " + csym2.ctype());
```

```
      ColumnSymbol csym = csym1.mergeWith(csym2);
      result.add(csym);
    }
    return new ColumnSymbols(result);
  }
  public int pos(Id id)
  {
    int i = 0;
    for (ColumnSymbol csym : list)
    {
      if (csym.id.name.equals(id.name)) return i;
      i++;
    }
    return -1;
  }
  public int pos(ColumnSymbol csym)
  {
    int i = 0;
    for (ColumnSymbol csym0 : list)
    {
      if (csym0 == csym) return i;
      i++;
    }
    return -1;
  }
}
public static record ColumnSymbolSet(Set<ColumnSymbol> set)
{
  public ColumnSymbolSet() { this(new HashSet<ColumnSymbol>()); }
  public ColumnSymbolSet add(ColumnSymbol... elems)
  {
    Set<ColumnSymbol> set0 = new HashSet<ColumnSymbol>(set);
    set0.addAll(Arrays.asList(elems));
    return new ColumnSymbolSet(set0);
  }
}
public static sealed interface ColumnType
{
  public Type type();
  public int size();
  public static record Null() implements ColumnType
  {
    public Type type() { throw new TypeError("cannot convert column type NULL to value type"); }
    public String toString() { return "NULL"; }
    public int size() { return 0; }
  }
  public static record Int(int n) implements ColumnType
  {
    public Type type() { return new Type.Int(); }
    public String toString() { return "INT"; }
    public int size() { return n; }
  }
  public static record VarChar(int n) implements ColumnType
  {
```

```
    public Type type() { return new Type.String(); }
    public String toString() { return "VARCHAR"; }
    public int size() { return n; }
  }
}
public static sealed interface ConstraintSymbol
{
  public static record Default(Value value) implements ConstraintSymbol { }
  public static record Notnull() implements ConstraintSymbol { }
  public static record Unique() implements ConstraintSymbol { }
}
public static record ConstraintSet(Set<ConstraintSymbol> set)
{
  public ConstraintSet() { this(new HashSet<ConstraintSymbol>()); }
  public ConstraintSet add(ConstraintSymbol... elems)
  {
    Set<ConstraintSymbol> set0 = new HashSet<ConstraintSymbol>(set);
    set0.addAll(Arrays.asList(elems));
    return new ConstraintSet(set0);
  }
  public Value getDefault()
  {
    for (ConstraintSymbol csym : set)
    {
      if (csym instanceof ConstraintSymbol.Default)
      {
        ConstraintSymbol.Default csym0 = (ConstraintSymbol.Default)csym;
        return csym0.value;
      }
    }
    return null;
  }
  public boolean hasNotnull()
  {
    for (ConstraintSymbol csym : set)
    {
      if (csym instanceof ConstraintSymbol.Notnull) return true;
    }
    return false;
  }
  public boolean hasUnique()
  {
    for (ConstraintSymbol csym : set)
    {
      if (csym instanceof ConstraintSymbol.Unique) return true;
    }
    return false;
  }
}

public static enum Kind { column, cell;
  public static Kind combine(Kind...kinds)
  {
    Kind result = null;
```

```
    for (Kind kind: kinds)
    {
      if (result == null) { result = kind; continue; }
      if (kind == column) { result = column; break; }
    }
    return result;
  }
}
public static record Kinds(List<Kind> list)
{
  public Kinds() { this(new ArrayList<Kind>()); }
}

public static record Id(String name) { }
public static record IdSeq(List<Id> list)
{
  public IdSeq() { this(new ArrayList<Id>()); }
  public IdSeq add(Id... elems)
  {
    List<Id> list0 = new ArrayList<Id>(list);
    list0.addAll(Arrays.asList(elems));
    return new IdSeq(list0);
  }
}
public static record IdSet(Set<Id> set)
{
  public IdSet() { this(new HashSet<Id>()); }
  public IdSet add(Id... elems)
  {
    Set<Id> set0 = new HashSet<Id>(set);
    set0.addAll(Arrays.asList(elems));
    return new IdSet(set0);
  }
}
public static record IdOption(Optional<Id> option) { }

public static record ExpType(Type type, Kind kind, boolean grouped)
{
  public ColumnSymbol columnSymbol()
  {
    ColumnType ctype = type.columnType();
    return new ColumnSymbol(new Id("_"), ctype,
        new ConstraintSet(), new IdOption(Optional.empty()));
  }
  public boolean matches(Type type1)
  {
    return type.matches(type1);
  }
  public boolean atomicType()
  {
    return type.atomicType();
  }
  public static ExpType combine(ExpType... etypes)
  {
```

```
    Type type = null;
    Kind kind = null;
    boolean grouped = true;
    for (ExpType etype0 : etypes)
    {
      type = Type.combine(type, etype0.type());
      kind = Kind.combine(kind, etype0.kind());
      grouped = grouped & etype0.grouped();
    }
    return new ExpType(type, kind, grouped);
  }
}
public static record ExpTypes(List<ExpType> list)
{
  public ExpTypes() { this(new ArrayList<ExpType>()); }
  public ExpTypes add(ExpType... elems)
  {
    List<ExpType> list0 = new ArrayList<ExpType>(list);
    list0.addAll(Arrays.asList(elems));
    return new ExpTypes(list0);
  }
  public ColumnSymbols columnSymbols()
  {
    List<ColumnSymbol> csyms = new ArrayList<ColumnSymbol>();
    for (ExpType etype : list)
      csyms.add(etype.columnSymbol());
    return new ColumnSymbols(csyms);
  }
}

public static boolean matchingAtomicTypes(Type... types)
{
  Type type = null;
  for (Type type0 : types)
  {
    if (type == null)
    {
      type = type0.elemType();
      if (type instanceof Type.Seq) return false;
      continue;
    }
    if (!type0.matches(type)) return false;
  }
  return true;
}
public static boolean matchingAtomicTypes(ExpType... etypes)
{
  Type type = null;
  for (ExpType etype : etypes)
  {
    if (type == null)
    {
      type = etype.type().elemType();
      if (type instanceof Type.Seq) return false;
```

```
      continue;
    }
    if (!etype.matches(type)) return false;
  }
  return true;
}

public static sealed interface Type
{
  public ColumnType columnType();
  public Type elemType();
  public boolean sameType(Type type);
  default public ExpType expType(ExpType etype1, ExpType etype2)
  {
    Kind kind = Kind.combine(etype1.kind, etype2.kind);
    boolean grouped = etype1.grouped() && etype2.grouped();
    return new ExpType(this, kind, grouped);
  }
  default public boolean atomicType()
  {
    Type type0 = this.elemType();
    return !(type0 instanceof Type.Seq);
  }
  default public boolean matches(Type type1)
  {
    Type type0 = this.elemType();
    if (type0 instanceof Type.Null) return true;
    type1 = type1.elemType();
    if (type1 instanceof Type.Null) return true;
    return type0.sameType(type1);
  }
  public static Type combine(Type... types)
  {
    Type result = null;
    for (Type type : types)
    {
      if (result == null || result instanceof Null)
        result = type.elemType();
    }
    return result;
  }
  public static record Null() implements Type
  {
    public ColumnType columnType() { return new ColumnType.Null(); }
    public Type elemType() { return this; }
    public java.lang.String toString() { return "NULL"; }
    public boolean sameType(Type type) { return type instanceof Null; }
  }
  public static record Int() implements Type
  {
    public ColumnType columnType() { return new ColumnType.Int(10); }
    public Type elemType() { return this; }
    public boolean sameType(Type type) { return type instanceof Int; }
    public java.lang.String toString() { return "INT"; }
```

```
  }
  public static record Bool() implements Type
  {
    public ColumnType columnType() { throw new TypeError("cannot convert Bool to column type"); }
    public Type elemType() { return this; }
    public boolean sameType(Type type) { return type instanceof Bool; }
    public java.lang.String toString() { return "BOOL"; }
  }
  public static record String() implements Type
  {
    public ColumnType columnType() { return new ColumnType.VarChar(10); }
    public Type elemType() { return this; }
    public boolean sameType(Type type) { return type instanceof String; }
    public java.lang.String toString() { return "STRING"; }
  }
  public static record Seq(List<Type> list) implements Type
  {
    public ColumnType columnType() { throw new TypeError("cannot convert Seq to column type"); }
    public Type elemType()
    {
      if (list.size() == 1) return list.get(0);
      return this;
    }
    public boolean sameType(Type type)
    {
      if (!(type instanceof Seq)) return false;
      Seq type0 = (Seq)type;
      int n = list.size();
      if (n != type0.list.size()) return false;
      for (int i = 0; i < n; i++)
      {
        Type t1 = list.get(i);
        Type t2 = type0.list.get(i);
        if (!t1.sameType(t2)) return false;
      }
      return true;
    }
    public java.lang.String toString()
    {
      StringBuffer buffer = new StringBuffer();
      buffer.append("SEQ(");
      int n = list.size();
      int i = 0;
      for (Type type : list)
      {
        buffer.append(type);
        i++;
        if (i < n) buffer.append(",");
      }
      buffer.append(")");
      return buffer.toString();
    }
  }
}
```

```
@SuppressWarnings("unchecked")
public static class Annotation
{
  private Object value;
  public Annotation() { value = null; }

  private static boolean show = true;
  public static void show(boolean show) { Annotation.show = show; };

  public String toString()
  {
    if (!show || value == null) return "";
    return "{" + value.toString() + "}";
  }

  private static class Pair<T1,T2>
  {
    public final T1 v1; public final T2 v2;
    public Pair(T1 v1, T2 v2) { this.v1 = v1; this.v2 = v2; }
    public String toString() { return v1 + "," + v2; }
  }
  /*
  private static class Triple<T1,T2,T3>
  {
    public final T1 v1; public final T2 v2; public final T3 v3;
    public Triple(T1 v1, T2 v2, T3 v3)
    { this.v1 = v1; this.v2 = v2; this.v3 = v3; }
    public String toString() { return v1 + "," + v2 + "," + v3; }
  }
  */
  private static class Quadrupel<T1,T2,T3,T4>
  {
    public final T1 v1; public final T2 v2;
    public final T3 v3; public final T4 v4;
    public Quadrupel(T1 v1, T2 v2, T3 v3, T4 v4)
    { this.v1 = v1; this.v2 = v2; this.v3 = v3; this.v4 = v4; }
    public String toString() { return v1 + "," + v2 + "," + v3 + "," + v4; }
  }

  public void setValue(Object value) { this.value = value; }
  public Object getValue() { return value; }

  public void setTableSymbol(TableSymbol tsym) { value = tsym; }
  public TableSymbol getTableSymbol() { return (TableSymbol)value; }

  public void setColumnSymbols(ColumnSymbols csyms) { value = csyms; }
  public ColumnSymbols getColumnSymbols() { return (ColumnSymbols)value; }

  public void setColumnSymbolsPair(ColumnSymbols csyms1, ColumnSymbols csyms2)
  { value = new Pair<ColumnSymbols,ColumnSymbols>(csyms1,csyms2); }
  public ColumnSymbols getColumnSymbols1()
  { return ((Pair<ColumnSymbols,ColumnSymbols>)value).v1; }
  public ColumnSymbols getColumnSymbols2()
```

```
  { return ((Pair<ColumnSymbols,ColumnSymbols>)value).v2; }

  public void setPosition(Integer pos) { value = pos; }
  public Integer getPosition() { return (Integer)value; }

  public void setPositionSeq(List<Integer> pos) { value = pos; }
  public List<Integer> getPositionSeq() { return (List<Integer>)value; }

  public void setTableInfo(TableSymbol tsym, ColumnSymbols csyms)
  { value = new Pair<TableSymbol,ColumnSymbols>(tsym,csyms); }
  public TableSymbol getTableInfo()
  { return ((Pair<TableSymbol,ColumnSymbols>)value).v1; }
  public ColumnSymbols getTableColumns()
  { return ((Pair<TableSymbol,ColumnSymbols>)value).v2; }

  public void setPositionSeqPair(List<Integer> pos1, List<Integer> pos2)
  { value = new Pair<List<Integer>,List<Integer>>(pos1,pos2); }
  public List<Integer> getPositionSeq1()
  { return ((Pair<List<Integer>,List<Integer>>)value).v1; }
  public List<Integer> getPositionSeq2()
  { return ((Pair<List<Integer>,List<Integer>>)value).v2; }

  public void setPosAsc(Integer pos, Boolean asc)
  { value = new Pair<Integer,Boolean>(pos,asc); }
  public Integer getPos()
  { return ((Pair<Integer,Boolean>)value).v1; }
  public Boolean getAsc()
  { return ((Pair<Integer,Boolean>)value).v2; }

  public void setPosAscSeq(List<Integer> pos, List<Boolean> asc)
  { value = new Pair<List<Integer>,List<Boolean>>(pos,asc); }
  public List<Integer> getPosSeq()
  { return ((Pair<List<Integer>,List<Boolean>>)value).v1; }
  public List<Boolean> getAscSeq()
  { return ((Pair<List<Integer>,List<Boolean>>)value).v2; }

  public void setColumnInfo(ColumnSymbol csym,
    Integer pos, Kind kind, Integer depth)
  { value = new Quadrupel<ColumnSymbol,Integer,Kind,Integer>
      (csym, pos, kind, depth); }
  public ColumnSymbol getColumnSymbol()
  { return ((Quadrupel<ColumnSymbol,Integer,Kind,Integer>)value).v1; }
  public Integer getColumnPos()
  { return ((Quadrupel<ColumnSymbol,Integer,Kind,Integer>)value).v2; }
  public Kind getColumnKind()
  { return ((Quadrupel<ColumnSymbol,Integer,Kind,Integer>)value).v3; }
  public Integer getColumnDepth()
  { return ((Quadrupel<ColumnSymbol,Integer,Kind,Integer>)value).v4; }

  public void setPattern(String regex)
  { value = pattern(regex); }
  public Pattern getPattern()
  { return (Pattern)value; }
}
```

```
// https://jku.academic-ai.at/general-chat, date 2025/02/07, prompt:
// "generate a java function called "pattern" which takes as argument a
// String object and returns a Pattern object. The content of the String
// object is a pattern as allowed by the syntax of the SQL LIKE operator.
// The Pattern object shall match any string allowed by the SQL pattern.
public static Pattern pattern(String sqlPattern) {
  // Escape all regex special characters in the input string
  String regex = sqlPattern
      .replace("\\", "\\\\")  // Escape backslashes
      .replace(".", "\\.")    // Escape dots
      .replace("?", "\\?")    // Escape question marks
      .replace("*", "\\*")    // Escape asterisks
      .replace("+", "\\+")    // Escape plus signs
      .replace("(", "\\(")    // Escape opening parentheses
      .replace(")", "\\)")    // Escape closing parentheses
      .replace("{", "\\{")    // Escape opening curly braces
      .replace("}", "\\}")    // Escape closing curly braces
      .replace("[", "\\[")    // Escape opening square brackets
      .replace("]", "\\]")    // Escape closing square brackets
      .replace("^", "\\^")    // Escape carets
      .replace("$", "\\$");   // Escape dollar signs

  // Replace SQL LIKE wildcards with regex equivalents
  regex = regex
      .replace("%", ".*")  // Replace % with .*
      .replace("_", ".");  // Replace _ with .

  // Compile the regex pattern
  return Pattern.compile(regex);
}

public static class TypeError extends RuntimeException
{
  public static final long serialVersionUID = 20241024L;
  public TypeError(String msg) { super(msg); }
}
public static void check(boolean cond, String msg)
{
  if (!cond) throw new TypeError(msg);
}

public static <T> boolean forall(Collection<T> values, Predicate<T> pred)
{
  for (T value : values)
  {
    if (!pred.test(value)) return false;
  }
  return true;
}

public static <T> boolean exists(Collection<T> values, Predicate<T> pred)
{
  for (T value : values)
```

```
    {
      if (pred.test(value)) return true;
    }
    return false;
  }

  public static <T> List<T> list(Collection<T> values, UnaryOperator<T> op)
  {
    List<T> list = new ArrayList<T>();
    for (T value : values) list.add(op.apply(value));
    return list;
  }

  public static <K,V> Map<K,V> map(Collection<K> keys, Function<K,V> fun)
  {
    Map<K,V> map = new HashMap<K,V>();
    for (K key : keys) map.put(key, fun.apply(key));
    return map;
  }
}
//---------------------------------------------------------------------------
// end of file
//---------------------------------------------------------------------------
```

B.3 Semantics Utilities

```
// ---------------------------------------------------------------------------
// TypeSystem.txt
// Auxiliaries for the SubSQL type checker.
// (c) 2024 https://www.risc.jku.at/research/formal/software/SLANG
// Wolfgang.Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <william.steingartner@tuke.sk>
// ---------------------------------------------------------------------------
package subsql;

import java.util.*;
import java.util.function.*;
import java.io.*;

import database.*;
import static subsql.TypeSystem.*;

public class Semantics
{
  public static record State(PrintStream out,
    Store store, Database db, Tables tables, Table rows)
  {
    public State(PrintStream out, Store store, Database db)
    { this(out, store, db, new Tables(), new Table()); }

    public State newTable(Table table)
    {
      List<Table> list = new ArrayList<Table>(tables.list);
```

```
    list.add(table);
    return new State(out, store, db, new Tables(list), rows);
  }
  public Table currentTable()
  {
    return tables.list.getLast();
  }
  public State newRow(Row row)
  {
    List<Row> list = new ArrayList<Row>(rows.list);
    list.add(row);
    return new State(out, store, db, tables, new Table(list));
  }
  public Row currentRow()
  {
    return rows.list.getLast();
  }
}

public static record Store(Map<Id,Value> map)
{
  public Store() { this(new HashMap<Id,Value>()); }
}
public static record Table(List<Row> list)
{
  public Table() { this(new ArrayList<Row>()); }
  public void println(PrintStream out, ColumnSymbols csyms)
  {
    List<Integer> sizes = null;
    if (csyms != null)
    {
      sizes = new ArrayList<Integer>();
      for (ColumnSymbol csym : csyms.list())
      {
        int size = csym.ctype().size();
        sizes.add(size);
        String text = pad(csym.id().name(), size);
        out.print(text);
        out.print('|');
      }
      out.println();
      for (Integer size : sizes)
      {
        out.print("-".repeat(size));
        out.print('+');
      }
      out.println();
    }
    for (Row row : list)
    {
      int i = 0;
      for (Value value : row.list())
      {
        String text = value.toString();
```

```
      if (sizes != null) text = pad(text, sizes.get(i));
      out.print(text);
      out.print('|');
      i++;
    }
    out.println();
  }
}
private static String pad(String text, int size)
{
  int len = text.length();
  if (len > size) return text.substring(0, size);
  return text + " ".repeat(size-len);
}
public String constraintViolation(TableSymbol tsym)
{
  int cnum = 0;
  for (ColumnSymbol csym : tsym.csyms().list())
  {
    for (ConstraintSymbol cons : csym.cons().set())
    {
      if (cons instanceof ConstraintSymbol.Notnull)
      {
        int rnum = 0;
        for (Row row : list)
        {
          Value value = row.list().get(cnum);
          if (value instanceof Value.Null)
            return "value " + value + " in row " + (rnum+1) +
                " and column " + (cnum+1) +
                " violates the NOT NULL (or PRIMARY KEY) constraint" +
                " declared for column " + csym.id().name() +
                " in table " + tsym.id().name();
          rnum++;
        }
      }
      else if (cons instanceof ConstraintSymbol.Unique)
      {
        int rlen = list.size();
        for (int i = 0; i < rlen; i++)
        {
          Value value1 = list.get(i).list.get(cnum);
          for (int j = i+1; j < rlen; j++)
          {
            Value value2 = list.get(j).list.get(cnum);
            if (value1.equals(value2))
              return "value " + value1 + " appears in rows " + (i+1) +
                  " and " + (j+1) + " in column " + (cnum+1) +
                  " which violates the UNIQUE (or PRIMARY KEY) constraint " +
                  " declared for column " + csym.id().name() +
                  " in table " + tsym.id().name();
          }
        }
      }
```

```
    }
    cnum++;
  }
  return null;
}
public Table filter(Predicate<Row> pred)
{
  List<Row> list0 = new ArrayList<Row>();
  for (Row row : list)
  {
    if (!pred.test(row)) continue;
    list0.add(row);
  }
  return new Table(list0);
}
public Table append(Table table)
{
  List<Row> list0 = new ArrayList<Row>(list);
  list0.addAll(table.list());
  return new Table(list0);
}
public Table crossJoin(Table table)
{
  List<Row> list0 = new ArrayList<Row>();
  for (Row row1 : list)
  {
    for (Row row2 : table.list())
    {
      list0.add(row1.append(row2));
    }
  }
  return new Table(list0);
}
public Table allRows() { return this; }
public Table distinctRows()
{
  List<Row> list0 = new ArrayList<Row>();
  for (Row row : list)
  {
    if (contains(list0, row)) continue;
    list0.add(row);
  }
  return new Table(list0);
}
private static boolean contains(List<Row> rows, Row row)
{
  for (Row row0 : rows)
  {
    if (row0.equals(row)) return true;
  }
  return false;
}
public boolean contains(Row row)
{
```

```
    return contains(list, row);
  }
  public Table sort(List<Integer> pos, List<Boolean> asc)
  {
    if (pos.isEmpty()) return this;
    List<Row> rows = new ArrayList<Row>(list);
    Collections.sort(rows, (Row row1, Row row2)->
    {
      int pn = pos.size();
      for (int i = 0; i < pn; i++)
      {
        int p = pos.get(i);
        Value value1 = row1.list().get(p);
        Value value2 = row2.list().get(p);
        if (value1.equals(value2)) continue;
        boolean ascending = asc.get(i);
        if (value1.less(value2))
        {
          return (ascending) ? -1 : +1;
        }
        else
        {
          return (ascending) ? +1 : -1;
        }
      }
      return 0;
    });
    return new Table(rows);
  }
  public Table tableRows(TableSymbol tsym, ColumnSymbols csyms)
  {
    List<Row> list0 = new ArrayList<Row>();
    for (Row row : list) list0.add(row.tableRow(tsym, csyms));
    return new Table(list0);
  }
  public Table select(List<Integer> is)
  {
    List<Row> list0 = new ArrayList<Row>();
    for (Integer i : is) list0.add(list.get(i));
    return new Table(list0);
  }
}
public static record Tables(List<Table> list)
{
  public Tables() { this(new ArrayList<Table>()); }
  public Tables append(Table table)
  {
    List<Table> list0 = new ArrayList<Table>(list);
    list0.add(table);
    return new Tables(list0);
  }
  public Tables appendAll(Tables tables)
  {
    List<Table> list0 = new ArrayList<Table>(list);
```

```
    list0.addAll(tables.list);
    return new Tables(list0);
  }
  public Tables filter(Predicate<Table> pred)
  {
    List<Table> list0 = new ArrayList<Table>();
    for (Table table : list)
    {
      if (!pred.test(table)) continue;
      list0.add(table);
    }
    return new Tables(list0);
  }
}

public static record Row(List<Value> list)
{
  public static Row nulls(int n)
  {
    List<Value> list = new ArrayList<Value>(n);
    for (int i = 0; i < n; i++) list.add(new Value.Null());
    return new Row(list);
  }
  public Row append(Row row)
  {
    List<Value> list0 = new ArrayList<Value>(list);
    list0.addAll(row.list);
    return new Row(list0);
  }
  public Row project(List<Integer> pos)
  {
    List<Value> result = new ArrayList<Value>(pos.size());
    for (Integer p : pos) result.add(list.get(p));
    return new Row(result);
  }
  public Row set(int pos, Value value)
  {
    List<Value> list0 = new ArrayList<Value>(list);
    list0.set(pos, value);
    return new Row(list0);
  }
  public Row tableRow(TableSymbol tsym, ColumnSymbols csyms)
  {
    List<Value> list0 = new ArrayList<Value>();
    for (ColumnSymbol tcsym : tsym.csyms().list())
    {
      int pos = csyms.pos(tcsym);
      Value value = null;
      if (pos != -1)
        value = list.get(pos);
      else
      {
        value = tcsym.cons().getDefault();
        if (value == null) value = new Value.Null();
```

```
      }
      list0.add(value);
    }
    return new Row(list0);
  }
}

public static sealed interface Value
{
  public boolean less(Value value);
  default public boolean lessEqual(Value value)
  {
    return equals(value) || less(value);
  }
  default public boolean isTrue()
  {
    check(this instanceof Bool, "value " + this + " is not a Boolean");
    Value.Bool value = (Value.Bool)this;
    return value.bool;
  }
  default Value isIn(Values values)
  {
    if (this instanceof Value.Null)
      return new Value.Null();
    if (values.exists((Value value0)->(value0.equals(this))))
      return new Value.Bool(true);
    if (values.exists((Value value0)->(value0 instanceof Value.Null)))
      return new Value.Null();
    return new Value.Bool(false);
  }
  public record Null() implements Value
  {
    public java.lang.String toString() { return "NULL"; }
    public boolean less(Value value)
    {
      switch (value)
      {
      case Null() -> { return false; }
      default -> { return true; }
      }
    }
  }
  public record Int(int integer) implements Value
  {
    public java.lang.String toString() { return Integer.toString(integer); }
    public boolean less(Value value)
    {
      switch (value)
      {
      case Null() -> { return false; }
      case Int(int integer0) -> { return integer < integer0; }
      default -> { throw new SemanticsError("comparison of value " + this +
         " with incompatible value " + value); }
      }
```

```
  }
}
public record Bool(boolean bool) implements Value
{
  public java.lang.String toString() { return Boolean.toString(bool); }
  public boolean less(Value value)
  {
    switch (value)
    {
    case Null() -> { return false; }
    case Bool(boolean bool0) -> { return !bool && bool0; }
    default -> { throw new SemanticsError("comparison of value " + this +
       " with incompatible value " + value); }
    }
  }
}
public record String(java.lang.String string) implements Value
{
  public java.lang.String toString() { return string; }
  public boolean less(Value value)
  {
    switch (value)
    {
    case Null() -> { return false; }
    case String(java.lang.String string0) ->
    { return string.compareTo(string0) < 0; }
    default -> { throw new SemanticsError("comparison of value " + this +
        " with incompatible value " + value); }
    }
  }
}
public record Seq(Values values) implements Value
{
  public java.lang.String toString() { return values.toString(); }
  public boolean less(Value value)
  {
    switch (value)
    {
    case Null() -> { return false; }
    case Seq(Values values0) ->
    {
      int len = values.list.size();
      int len0 = values0.list.size();
      if (len != len0)
        throw new SemanticsError("comparison of sequence " + this +
            " of length " + len + " with sequence " + value +
            " of length " + len0);
      for (int i = 0; i < len; i++)
      {
        Value value1 = values.list.get(i);
        Value value2 = values0.list.get(i);
        if (value1.equals(value2)) continue;
        return value1.less(value2);
      }
```

```
        return false;
      }
      default -> { throw new SemanticsError("comparison of value " + this +
          " with incompatible value " + value);
      }
      }
    }
  }
}
public static record Values(List<Value> list)
{
  @SuppressWarnings("unchecked")
  public boolean equals(Object object)
  {
    if (!(object instanceof List)) return false;
    List<Value> list0 = (List<Value>)object;
    return list.equals(list0);
  }
  public int hashCode() { return list.hashCode(); }
  public boolean exists(Predicate<Value> pred)
  {
    for (Value value : list)
    {
      if (pred.test(value)) return true;
    }
    return false;
  }
}

public static record ValueOption(Optional<Value> optional) { }

public static Value intFunction(State s,
  Function<State,Value> e1, Function<State,Value> e2,
  BiFunction<Integer,Integer,Integer> fun)
{
  Value value1 = e1.apply(s);
  Value value2 = e2.apply(s);
  if (value1 instanceof Value.Null) return new Value.Null();
  if (value2 instanceof Value.Null) return new Value.Null();
  check(value1 instanceof Value.Int, "value " + value1 + " is not an integer");
  check(value2 instanceof Value.Int, "value " + value2 + " is not an integer");
  Value.Int integer1 = (Value.Int)value1;
  Value.Int integer2 = (Value.Int)value2;
  return new Value.Int(fun.apply(integer1.integer, integer2.integer));
}

public static Value valuePred(State s,
  Function<State,Value> e1, Function<State,Value> e2,
  BiPredicate<Value,Value> pred)
{
  Value value1 = e1.apply(s);
  Value value2 = e2.apply(s);
  if (value1 instanceof Value.Null) return new Value.Null();
  if (value2 instanceof Value.Null) return new Value.Null();
```

```
  return new Value.Bool(pred.test(value1, value2));
}

public static Value binaryAnd(State s,
  Function<State,Value> e1, Function<State,Value> e2)
{
  Value v1 = e1.apply(s);
  return switch (v1) {
          case Value.Bool(var b1) ->
            (b1 == false) ? new Value.Bool(false) : e2.apply(s);
          default ->
            switch (e2.apply(s)) {
              case Value.Bool(var b2) ->
                (b2 == false) ? new Value.Bool(false) : v1;
              default -> new Value.Null();
            };
      };
}

public static Value binaryOr(State s,
  Function<State,Value> e1, Function<State,Value> e2)
{
  Value v1 = e1.apply(s);
  return switch (v1) {
          case Value.Bool(var b1) ->
            (b1 == true) ? new Value.Bool(true) : e2.apply(s);
          default ->
            switch (e2.apply(s)) {
              case Value.Bool(var b2) ->
                (b2 == true) ? new Value.Bool(true) : v1;
              default -> new Value.Null();
            };
      };
}

public static Table crossJoin(Table table1, Table table2,
  Predicate<Row> pred)
{
  return table1.crossJoin(table2).filter(pred);
}

public static Table leftJoin(Table table1, Table table2, int n2,
  Predicate<Row> pred)
{
  List<Row> list = new ArrayList<Row>();
  for (Row row1 : table1.list())
  {
    List<Row> rows1 = new ArrayList<Row>();
    for (Row row2 : table2.list()) rows1.add(row1.append(row2));
    List<Row> rows2 = new Table(rows1).filter(pred).list();
    if (rows2.isEmpty()) rows2.add(row1.append(Row.nulls(n2)));
    list.addAll(rows2);
  }
  return new Table(list);
```

```
  }

  public static Table rightJoin(Table table1, int n1, Table table2,
    Predicate<Row> pred)
  {
    List<Row> list = new ArrayList<Row>();
    for (Row row2 : table2.list())
    {
      List<Row> rows1 = new ArrayList<Row>();
      for (Row row1 : table1.list()) rows1.add(row1.append(row2));
      List<Row> rows2 = new Table(rows1).filter(pred).list();
      if (rows2.isEmpty()) rows2.add(Row.nulls(n1).append(row2));
      list.addAll(rows2);
    }
    return new Table(list);
  }

  public static Table fullJoin(Table table1, int n1, Table table2, int n2,
    Predicate<Row> pred)
  {
    List<Row> list = new ArrayList<Row>(crossJoin(table1, table2, pred).list());
    for (Row row1 : table1.list())
    {
      List<Row> rows1 = new ArrayList<Row>();
      for (Row row2 : table2.list()) rows1.add(row1.append(row2));
      List<Row> rows2 = new Table(rows1).filter(pred).list();
      if (!rows2.isEmpty()) continue;
      rows2.add(row1.append(Row.nulls(n2)));
      list.addAll(rows2);
    }
    for (Row row1 : table1.list())
    {
      List<Row> rows1 = new ArrayList<Row>();
      for (Row row2 : table2.list()) rows1.add(row1.append(row2));
      List<Row> rows2 = new Table(rows1).filter(pred).list();
      if (!rows2.isEmpty()) continue;
      rows2.add(row1.append(Row.nulls(n2)));
      list.addAll(rows2);
    }
    return new Table(list);
  }

  public static class SemanticsError extends RuntimeException
  {
    public static final long serialVersionUID = 20250206L;
    public SemanticsError(String msg) { super(msg); }
  }
  public static void check(boolean cond, String msg)
  {
    if (!cond) throw new SemanticsError(msg);
  }
}
// ---------------------------------------------------------------------------
// end of file
```

```
// ----------------------------------------------------------------------
```

B.4 Database

```
// ----------------------------------------------------------------------------
// Database.java
// A simple database interface.
// (c) 2024, Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// ----------------------------------------------------------------------------
package database;

import java.io.*;
import java.util.*;

public interface Database
{
  // read/write database from/to file
  public void read(String path) throws IOException;
  public void write(String path) throws IOException;

  // create table with typed columns and drop it
  public void createTable(String name, Info info);
  public void dropTable(String name);

  // get names of tables in database
  public Set<String> getTableNames();

  // get information for named table
  public Info getInfo(String name);
  public record Info(List<Column> columns) { }
  public record Column(String name, Type type, Value dvalue,
    boolean notnull, boolean unique)
  { }
  public sealed interface Type permits Type.Int, Type.VarChar
  {
    public record Int(int n) implements Type { }
    public record VarChar(int n) implements Type { }
  }

  // get and set content of named table
  public Content getContent(String name);
  public void setContent(String name, Content content);
  public record Content(List<Row> rows) { }
  public record Row(List<Value> values) { }
  public sealed interface Value permits Value.Int, Value.VarChar, Value.Null
  {
    public record Int(int i) implements Value { }
    public record VarChar(String s) implements Value { }
    public record Null() implements Value { }
  }
}
//----------------------------------------------------------------------------
// end of file
```

```
//---------------------------------------------------------------------------

// ---------------------------------------------------------------------------
// DatabaseClass.java
// A simple database implementation
// (c) 2024, Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// ---------------------------------------------------------------------------
package database;

import java.io.*;
import java.util.*;
import java.util.function.*;

import com.opencsv.*;
import com.opencsv.exceptions.*;

public class DatabaseClass implements Database
{
  // maps of table names to their information and contents
  private Map<String,Info> infos = new HashMap<String,Info>();
  private Map<String,Content> contents = new HashMap<String,Content>();

  @Override
  public Set<String> getTableNames()
  {
    return new TreeSet<String>(infos.keySet());
  }

  @Override
  public void createTable(String name, Info info)
  {
    List<Column> columns = Collections.unmodifiableList(info.columns());
    infos.put(name, new Info(columns));
    List<Row> rows = Collections.unmodifiableList(new ArrayList<Row>());
    contents.put(name, new Content(rows));
  }

  @Override
  public void dropTable(String name)
  {
    infos.remove(name);
    contents.remove(name);
  }

  @Override
  public Info getInfo(String name)
  {
    return infos.get(name);
  }

  @Override
  public Content getContent(String name)
  {
    return contents.get(name);
```

```
}

@Override
public void setContent(String name, Content content)
{
  // store unmodifiable copy that may be safely retrieved
  List<Row> rows = new ArrayList<Row>();
  for (Row row : content.rows())
  {
    List<Value> values = Collections.unmodifiableList(row.values());
    rows.add(new Row(values));
  }
  contents.put(name, new Content(rows));
}

/*****************************************************************************
 * Read content of database from file with given path.
 *****************************************************************************/
@Override
public void read(String path) throws IOException
{
  infos.clear();
  contents.clear();
  try (BufferedReader reader = new BufferedReader(new FileReader(path)))
  {
    while (true)
    {
      String table = reader.readLine();
      if (table == null) break;
      String string = readTable(reader);
      processTable(table, string);
    }
  }
}

/*****************************************************************************
 * Read string representation of table from database.
 *****************************************************************************/
private String readTable(BufferedReader reader) throws IOException
{
  String delim = readString(reader);
  StringBuffer buffer = new StringBuffer();
  while (true)
  {
    String line = readString(reader);
    if (line.equals(delim)) break;
    buffer.append(line);
    buffer.append("\n");
  }
  return buffer.toString();
}

/*****************************************************************************
 * Process string representation of denoted table.
```

```
 ***************************************************************************/
private void processTable(String table, String string) throws IOException
{
  try (CSVReader csvReader = new CSVReader(new StringReader(string)))
  {
    String[] names = readRow(csvReader, -1, "no column names", null);
    int n = names.length;
    String[] types = readRow(csvReader, n, "no column types", "wrong number of column types");
    String[] sizes = readRow(csvReader, n, "no column sizes", "wrong number of column sizes");
    String[] dvalues = readRow(csvReader, n, "no default values", "wrong number of default values");
    String[] nonnulls = readRow(csvReader, n, "no nonnull tags", "wrong number of nonnull tags");
    String[] uniques = readRow(csvReader, n, "no unique tags", "wrong number of unique tags");
    List<Column> columns = new ArrayList<Column>();
    for (int i = 0; i < n; i++)
    {
      String name = names[i];
      int size = parseInt(sizes[i]);
      Type type = columnType(types[i], size);
      Value dvalue = parseValue(dvalues[i], type);
      boolean nonnull = parseBoolean(nonnulls[i]);
      boolean unique = parseBoolean(uniques[i]);
      Column column = new Column(name, type, dvalue, nonnull, unique);
      columns.add(column);
    }
    Info info = new Info(columns);
    infos.put(table, info);
    List<Row> rows = new ArrayList<Row>();
    while (true)
    {
      String[] cells = readRow(csvReader, n, null, "wrong number of cells in row");
      if (cells == null) break;
      List<Value> values = new ArrayList<Value>();
      int i = 0;
      for (String cell : cells)
      {
        Value value = parseValue(cell, columns.get(i).type());
        values.add(value);
        i++;
      }
      Row row = new Row(values);
      rows.add(row);
    }
    Content  content = new Content(rows);
    contents.put(table, content);
  }
}

/***************************************************************************
 * Read row from reader and return it as row of n strings.
 * If nullMsg is not null and/or lengthMsg is not null, throw an exception
 * if the row read is null and/or does not have n entries, respectively.
 * If nullMsg is null and the row read is null, return null.
 ***************************************************************************/
private String[] readRow(CSVReader reader, int n,
```

```
  String nullMsg, String lengthMsg) throws IOException
{
  String[] row;
  try
  {
    row = reader.readNext();
  }
  catch(CsvValidationException e)
  {
    throw new IOException(e.getMessage());
  }
  if (row == null)
  {
    if (nullMsg != null) throw new IOException(nullMsg);
  }
  else if (row.length != n)
  {
    if (lengthMsg != null) throw new IOException(lengthMsg +
          " (" + row.length + " given, " + n + " expected)");
  }
  return row;
}

/**************************************************************************
 * Read string, throw exception, if there is none.
 **************************************************************************/
private static String readString(BufferedReader reader) throws IOException
{
  String line = reader.readLine();
  if (line == null) throw new IOException("unexpected end of file");
  return line;
}

/**************************************************************************
 * Parse integer from string, throw exception, if there is none.
 **************************************************************************/
private static int parseInt(String string) throws IOException
{
  try
  {
    return Integer.parseInt(string);
  }
  catch (NumberFormatException e)
  {
    throw new IOException(e.getMessage());
  }
}

/**************************************************************************
 * Parse boolean from string, throw exception, if there is none.
 **************************************************************************/
private static boolean parseBoolean(String string) throws IOException
{
  try
```

```
  {
    return Boolean.parseBoolean(string);
  }
  catch (NumberFormatException e)
  {
    throw new IOException(e.getMessage());
  }
}

/*****************************************************************************
 * Parse value of denoted type from string, throw exception, if there is none.
 *****************************************************************************/
private static Value parseValue(String string, Type type) throws IOException
{
  if (string.equals(""))
  {
    return new Value.Null();
  }
  if (type instanceof Type.VarChar)
  {
    return new Value.VarChar(new String(string));
  }
  try
  {
    return new Value.Int(Integer.parseInt(string));
  }
  catch (NumberFormatException e)
  {
    throw new IOException(e.getMessage());
  }
}

/*****************************************************************************
 * Write content of database to file with given path.
 *****************************************************************************/
@Override
public void write(String path) throws IOException
{
  try (PrintWriter writer = new PrintWriter(path))
  {
    for (String table : getTableNames())
    {
      writer.println(table);
      String csv = csvTable(table);
      String delim = delimiter(csv);
      writer.println(delim);
      writer.print(csv);
      writer.println(delim);
    }
  }
}

/*****************************************************************************
 * Get delimiter string not contained in given text.
```

```
 ****************************************************************************/
private String delimiter(String text)
{
  String delimiter = "***";
  while (text.contains(delimiter))
    delimiter += "*";
  return delimiter;
}

/****************************************************************************
 * Get CSV representation of table with denoted name.
 ****************************************************************************/
private String csvTable(String table) throws IOException
{
  Info info = infos.get(table);
  int n = info.columns().size();
  StringWriter swriter = new StringWriter();
  try (CSVWriter cwriter = new CSVWriter(swriter))
  {
    String[] names = getRow(n,
        (Integer i)->info.columns().get(i).name());
    cwriter.writeNext(names, false);
    String[] types = getRow(n,
        (Integer i)->typeName(info.columns().get(i).type()));
    cwriter.writeNext(types, false);
    String[] sizes = getRow(n,
        (Integer i)->Integer.toString(typeSize(info.columns().get(i).type())));
    cwriter.writeNext(sizes, false);
    String[] dvalues = getRow(n,
        (Integer i)->valueString(info.columns().get(i).dvalue()));
    cwriter.writeNext(dvalues, false);
    String[] notnulls = getRow(n,
        (Integer i)->Boolean.toString(info.columns().get(i).notnull()));
    cwriter.writeNext(notnulls, false);
    String[] uniques = getRow(n,
        (Integer i)->Boolean.toString(info.columns().get(i).unique()));
    cwriter.writeNext(uniques, false);
    Content content = contents.get(table);
    for (Row row : content.rows())
    {
      String[] cells = getRow(n,
          (Integer i)->valueString(row.values().get(i)));
      cwriter.writeNext(cells, false);
    }
  }
  return swriter.toString();
}

/****************************************************************************
 * Get row of n strings derived by application of given function.
 ****************************************************************************/
private String[] getRow(int n, Function<Integer,String> fun)
{
  String[] row = new String[n];
```

```
  for (int i = 0; i < n; i++)
    row[i] = fun.apply(i);
  return row;
}

/**************************************************************************
 * Get name of denoted type.
 **************************************************************************/
private static String typeName(Type type)
{
  switch (type)
  {
  case Type.Int(int size) -> { return "INT"; }
  case Type.VarChar(int size) -> { return "VARCHAR"; }
  }
}

/**************************************************************************
 * Get column type for denoted name and size, throw exception if none
 **************************************************************************/
private Type columnType(String name, int size) throws IOException
{
  switch(name)
  {
  case "INT" -> { return new Type.Int(size); }
  case "VARCHAR" -> { return new Type.VarChar(size); }
  default -> { throw new IOException("unknown type " + name); }
  }
}

/**************************************************************************
 * Get size of denoted type.
 **************************************************************************/
private static int typeSize(Type type)
{
  switch (type)
  {
  case Type.Int(int n) -> { return n; }
  case Type.VarChar(int n) -> { return n; }
  }
}

/**************************************************************************
 * Get string representation of denoted value
 **************************************************************************/
private static String valueString(Value value)
{
  if (value == null) return "";
  switch (value)
  {
  case Value.Int(int i) -> { return Integer.toString(i); }
  case Value.VarChar(String s) -> { return s; }
  case Value.Null() -> { return ""; }
  }
```

```
  }
}
// ---------------------------------------------------------------------------
// end of file
// ---------------------------------------------------------------------------

package database;

import java.util.*;

import static database.Database.*;
import static database.Database.Row;
import static database.Database.Value;
import static database.Database.Type;
import subsql.TypeSystem.*;
import subsql.Semantics.*;

public class DatabaseUtils
{
  public static Info info(ColumnSymbols csyms)
  {
    List<Column> columns = new ArrayList<Column>();
    for (ColumnSymbol csym : csyms.list())
    {
      String name = csym.id().name();
      Type type = type(csym.ctype());
      Value dvalue = value(csym.cons().getDefault());
      boolean notnull = csym.cons().hasNotnull();
      boolean unique = csym.cons().hasUnique();
      columns.add(new Column(name, type, dvalue, notnull, unique));
    }
    return new Info(columns);
  }

  public static Type type(ColumnType ctype)
  {
    switch (ctype)
    {
    case ColumnType.Int(int n) -> { return new Type.Int(n); }
    case ColumnType.VarChar(int n) -> { return new Type.VarChar(n); }
    default -> { return null; }
    }
  }

  public static Value value(subsql.Semantics.Value value)
  {
    if (value == null) return null;
    switch (value)
    {
    case subsql.Semantics.Value.Int(int n) -> { return new Value.Int(n); }
    case subsql.Semantics.Value.String(String s) -> { return new Value.VarChar(s); }
    default -> { return null; }
    }
  }
```

```
public static subsql.Semantics.Value value(Value value)
{
  switch (value)
  {
  case Value.Int(int n) -> { return new subsql.Semantics.Value.Int(n); }
  case Value.VarChar(String s) -> { return new subsql.Semantics.Value.String(s); }
  default -> { return null; }
  }
}

public static Table table(Content content)
{
  List<subsql.Semantics.Row> rows = new ArrayList<subsql.Semantics.Row>();
  for (Row row : content.rows())
  {
    List<subsql.Semantics.Value> values = new ArrayList<subsql.Semantics.Value>();
    for (Value value : row.values()) values.add(value(value));
    rows.add(new subsql.Semantics.Row(values));
  }
  return new Table(rows);
}

public static Content content(Table table)
{
  List<Row> rows = new ArrayList<Row>();
  for (subsql.Semantics.Row row : table.list())
  {
    List<Value> values = new ArrayList<Value>();
    for (subsql.Semantics.Value value : row.list()) values.add(value(value));
    rows.add(new Row(values));
  }
  return new Content(rows);
}

public static Env newEnv(Store store, Database db)
{
  Env env = new Env();
  for (Id id : store.map().keySet())
  {
    subsql.Semantics.Value value = store.map().get(id);
    if (env.venv().map().containsKey(id))
      throw new TypeError("double declaration of variable " + id.name());
    VarType type;
    if (value instanceof subsql.Semantics.Value.Int)
      type = VarType.intvar;
    if (value instanceof subsql.Semantics.Value.Bool)
      type = VarType.boolvar;
    else if (value instanceof subsql.Semantics.Value.String)
      type = VarType.stringvar;
    else
      throw new TypeError("illegal variable value " + value);
    env.venv().map().put(id, new VarSymbol(id, type));
  }
```

```
    for (String name : db.getTableNames())
    {
      Info info = db.getInfo(name);
      Id id = new Id(name);
      if (env.db().map().containsKey(id))
        throw new TypeError("double declaration of table " + id.name());
      ColumnSymbols csyms = csyms(id, info);
      env.db().map().put(id, new TableSymbol(id, csyms));
    }
    return env;
  }

  public static ColumnSymbols csyms(Id tid, Info info)
  {
    List<ColumnSymbol> csyms = new ArrayList<ColumnSymbol>();
    for (Column column : info.columns())
    {
      String name = column.name();
      Id id = new Id(name);
      ColumnType ctype = ctype(column.type());
      // currently, in db no constraints are stored
      ConstraintSet cset = new ConstraintSet();
      IdOption idopt = new IdOption(Optional.of(tid));
      ColumnSymbol csym = new ColumnSymbol(id, ctype, cset, idopt);
      csyms.add(csym);
    }
    return new ColumnSymbols(csyms);
  }

  public static ColumnType ctype(Type type)
  {
    switch (type)
    {
    case Type.Int(int n) -> { return new ColumnType.Int(n); }
    case Type.VarChar(int n) -> { return new ColumnType.VarChar(n); }
    default -> { return null; }
    }
  }
}
```

B.5 Application Programming Interface

```
// ----------------------------------------------------------------------------
// SubSQL.java
// (c) 2025, API of the SubSQL database language.
// Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// William Steingartner <William.Steingartner@tuke.sk>
// ----------------------------------------------------------------------------
package subsql;

import java.io.*;
import java.util.*;
import lang.subsql.SubSQL_parser;
```

```
public class SubSQL
{
  private PrintStream out;
  private Semantics.Store store;
  private database.Database db;

  private TypeSystem.Env env;
  private Semantics.State state;

  // SubSQL <sqlpath> [ <dbpath> ]
  public static void main(String[] args)
  {
    try
    {
      if (args.length < 1 || args.length > 2) return;
      String sqlpath = args[0];
      String dbpath = args.length == 2 ? args[1] : null;

      // the execution context
      SubSQL sql = new SubSQL();

      // reading the database from file
      if (dbpath != null) sql.read(dbpath);

      // demonstrate how to use meta-variables
      sql.set("NAME", "Wolfgang");

      // parsing, type-checking, executing the SubSQL commands in file
      sql.execute(new File(sqlpath));

      // demonstrate how to execute individual commands
      // Semantics.Table table = sql.execute(
      //   "SELECT * FROM Persons WHERE FirstName = {NAME};");
      // if (table != null) table.println(new PrintStream(System.out), null);

      // writing the database back to file
      if (dbpath != null) sql.write(dbpath);
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  public SubSQL()
  {
    this(new PrintStream(System.out),
        new Semantics.Store(), new database.DatabaseClass());
  }

  public SubSQL(PrintStream out, Semantics.Store store, database.Database db)
  {
    this.out = out;
```

```
  this.store = store;
  this.db = db;
  init();
}

public void init()
{
  this.env = database.DatabaseUtils.newEnv(store, db);
  this.state = new Semantics.State(out, store, db);
}

public void read(String path) throws IOException
{
  db.read(path);
  init();
}

public void write(String path) throws IOException
{
  db.write(path);
}

public void set(String var, String value)
{
  set(var, new Semantics.Value.String(value));
}

public void set(String var, Semantics.Value value)
{
  store.map().put(new TypeSystem.Id(var), value);
  init();
}

public List<Semantics.Table> execute(File file) throws FileNotFoundException
{
  lang.subsql.Session_._Operation operation = session(file);
  return execute(operation);
}

public lang.subsql.Session_._Operation session(File file) throws FileNotFoundException
{
  lang.subsql.Session stat = SubSQL_parser.parseSession(file);
  env = lang.subsql.Session_session.operation(stat).apply(env);
  return lang.subsql.Session_.operation(stat);
}

public List<Semantics.Table> execute(lang.subsql.Session_._Operation session)
{
  lang.subsql.Session_._Operation.Result result = session.apply(state);
  state = result._1();
  return result._2().list();
}

public Semantics.Table execute(String text)
```

```
  {
    lang.subsql.Stat_._Operation operation = stat(text);
    return execute(operation);
  }

  public lang.subsql.Stat_._Operation stat(String text)
  {
    lang.subsql.Stat stat = SubSQL_parser.parseStat(text);
    env = lang.subsql.Stat_stat.operation(stat).apply(env);
    return lang.subsql.Stat_.operation(stat);
  }

  public Semantics.Table execute(lang.subsql.Stat_._Operation stat)
  {
    lang.subsql.Stat_._Operation.Result result = stat.apply(state);
    state = result._1();
    if (result._2().list().isEmpty()) return null;
    return result._2().list().getLast();
  }
}
// ------------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------------
```

B.6 A Sample Execution

Input

```
CREATE TABLE Persons (
  PersonId INT(3) PRIMARY KEY,
  LastName VARCHAR(10) NOT NULL,
  FirstName VARCHAR(10) NOT NULL,
  Address VARCHAR(20)
);

INSERT INTO Persons VALUES (1, "Schreiner", "Wolfgang", "Linz");
INSERT INTO Persons VALUES (2, "Huber", "Wolfgang", "Linz");
INSERT INTO Persons VALUES (3, "Schreiner", "Thomas", "Linz");
INSERT INTO Persons VALUES (4, "Huber", "Thomas", "Linz");
INSERT INTO Persons VALUES (5, "Maier", "Thomas", "Linz");
INSERT INTO Persons VALUES (6, "Schmitt", "Thomas", "Linz");

SELECT * FROM Persons;

SELECT * FROM Persons WHERE FirstName = {NAME};

UPDATE Persons SET PersonId = PersonId+1;

SELECT * FROM Persons;

DELETE FROM Persons WHERE PersonId = 5;

SELECT * FROM Persons;
```

```
SELECT Address,PersonId,FirstName,LastName,LastName FROM Persons;

SELECT (PersonId+PersonId) FROM Persons;

CREATE TABLE Articles (
  ArticleId INT(3) PRIMARY KEY,
  Name VARCHAR(10) NOT NULL,
  Price INT(5) NOT NULL
);

INSERT INTO Articles VALUES (1, "Book", 10);
INSERT INTO Articles VALUES (2, "CD", 20);
INSERT INTO Articles VALUES (3, "DVD", 30);

SELECT * FROM Articles;

CREATE TABLE Sales (
  PersonId INT(3),
  ArticleId INT(3)
);

INSERT INTO Sales VALUES (2,1);
INSERT INTO Sales VALUES (2,2);
INSERT INTO Sales VALUES (3,1);
INSERT INTO Sales VALUES (3,2);
INSERT INTO Sales VALUES (3,3);
INSERT INTO Sales VALUES (4,2);

SELECT * FROM Sales;

SELECT A.Name, P.FirstName, P.LastName
  FROM Persons P, Articles A, Sales S
  WHERE P.PersonId = S.PersonId AND A.ArticleId = S.ArticleId
  ORDER BY A.Name;

SELECT COUNT(*) FROM Persons;
SELECT FirstName,COUNT(LastName) FROM Persons;
SELECT FirstName,COUNT(LastName) FROM Persons GROUP BY FirstName;
SELECT FirstName,COUNT(LastName) FROM Persons GROUP BY FirstName
  HAVING FirstName="Wolfgang";

SELECT * FROM Persons WHERE FirstName LIKE "%olf%";

SELECT *
  FROM Persons JOIN Sales ON (Persons.PersonId = Sales.PersonId);

SELECT *
  FROM Persons JOIN Sales USING (PersonId);

SELECT *
  FROM Persons JOIN Sales ON (Persons.PersonId = Sales.PersonId)
       JOIN Articles ON (Sales.ArticleId = Articles.ArticleId);
```

```
SELECT *
  FROM Persons JOIN Sales USING (PersonId) JOIN Articles USING (ArticleId);

CREATE TABLE Numbers (Number INT(10));
INSERT INTO Numbers VALUES(1);
INSERT INTO Numbers VALUES(2);
INSERT INTO Numbers VALUES(2);
INSERT INTO Numbers VALUES(3);

SELECT * FROM Numbers;
SELECT DISTINCT * FROM Numbers;

(SELECT * FROM Numbers) UNION DISTINCT (SELECT * FROM Numbers);
(SELECT * FROM Numbers) UNION ALL (SELECT * FROM Numbers);
(SELECT * FROM Numbers) INTERSECT (SELECT * FROM Numbers);
(SELECT * FROM Numbers) EXCEPT (SELECT * FROM Numbers);

SELECT * FROM Persons, Sales, Articles WHERE
  Persons.PersonId = Sales.PersonId AND
  Sales.ArticleId = Articles.ArticleId AND
  Persons.PersonId = (SELECT MIN(PersonId) FROM Persons);

SELECT * FROM Persons, Sales WHERE
  Persons.PersonId = (SELECT ArticleId+MIN(PersonId) FROM Persons);
```

Output

```
CREATE TABLE Persons(PersonId INT(3) PRIMARY KEY,LastName VARCHAR(10) NOT NULL,
  FirstName VARCHAR(10) NOT NULL,Address VARCHAR(20));
INSERT INTO Persons VALUES (1,"Schreiner","Wolfgang","Linz");
INSERT INTO Persons VALUES (2,"Huber","Wolfgang","Linz");
INSERT INTO Persons VALUES (3,"Schreiner","Thomas","Linz");
INSERT INTO Persons VALUES (4,"Huber","Thomas","Linz");
INSERT INTO Persons VALUES (5,"Maier","Thomas","Linz");
INSERT INTO Persons VALUES (6,"Schmitt","Thomas","Linz");
SELECT * FROM Persons;
Per|LastName  |FirstName |Address             |
---+----------+----------+--------------------+
1  |Schreiner |Wolfgang  |Linz                |
2  |Huber     |Wolfgang  |Linz                |
3  |Schreiner |Thomas    |Linz                |
4  |Huber     |Thomas    |Linz                |
5  |Maier     |Thomas    |Linz                |
6  |Schmitt   |Thomas    |Linz                |
SELECT * FROM Persons WHERE (FirstName = {NAME});
Per|LastName  |FirstName |Address             |
---+----------+----------+--------------------+
1  |Schreiner |Wolfgang  |Linz                |
2  |Huber     |Wolfgang  |Linz                |
UPDATE Persons SET PersonId = (PersonId+1);
SELECT * FROM Persons;
Per|LastName  |FirstName |Address             |
---+----------+----------+--------------------+
```

```
2  |Schreiner |Wolfgang  |Linz               |
3  |Huber     |Wolfgang  |Linz               |
4  |Schreiner |Thomas    |Linz               |
5  |Huber     |Thomas    |Linz               |
6  |Maier     |Thomas    |Linz               |
7  |Schmitt   |Thomas    |Linz               |
DELETE FROM Persons WHERE (PersonId = 5);
SELECT * FROM Persons;
Per|LastName  |FirstName |Address            |
---+----------+----------+-------------------+
2  |Schreiner |Wolfgang  |Linz               |
3  |Huber     |Wolfgang  |Linz               |
4  |Schreiner |Thomas    |Linz               |
6  |Maier     |Thomas    |Linz               |
7  |Schmitt   |Thomas    |Linz               |
SELECT Address,PersonId,FirstName,LastName,LastName FROM Persons;
Address             |Per|FirstName |LastName  |LastName  |
--------------------+---+----------+----------+----------+
Linz                |2  |Wolfgang  |Schreiner |Schreiner |
Linz                |3  |Wolfgang  |Huber     |Huber     |
Linz                |4  |Thomas    |Schreiner |Schreiner |
Linz                |6  |Thomas    |Maier     |Maier     |
Linz                |7  |Thomas    |Schmitt   |Schmitt   |
SELECT ((PersonId+PersonId)) FROM Persons;
_         |
----------+
4         |
6         |
8         |
12        |
14        |
CREATE TABLE Articles(ArticleId INT(3) PRIMARY KEY,Name VARCHAR(10) NOT NULL,
  Price INT(5) NOT NULL);
INSERT INTO Articles VALUES (1,"Book",10);
INSERT INTO Articles VALUES (2,"CD",20);
INSERT INTO Articles VALUES (3,"DVD",30);
SELECT * FROM Articles;
Art|Name      |Price|
---+----------+-----+
1  |Book      |10   |
2  |CD        |20   |
3  |DVD       |30   |
CREATE TABLE Sales(PersonId INT(3),ArticleId INT(3));
INSERT INTO Sales VALUES (2,1);
INSERT INTO Sales VALUES (2,2);
INSERT INTO Sales VALUES (3,1);
INSERT INTO Sales VALUES (3,2);
INSERT INTO Sales VALUES (3,3);
INSERT INTO Sales VALUES (4,2);
SELECT * FROM Sales;
Per|Art|
---+---+
2  |1  |
2  |2  |
```

```
3  |1  |
3  |2  |
3  |3  |
4  |2  |
SELECT A.Name,P.FirstName,P.LastName FROM Persons P,Articles A,Sales S WHERE
  ((P.PersonId = S.PersonId) AND (A.ArticleId = S.ArticleId)) ORDER BY A.Name;
Name      |FirstName |LastName  |
----------+----------+----------+
Book      |Wolfgang  |Schreiner |
Book      |Wolfgang  |Huber     |
CD        |Wolfgang  |Schreiner |
CD        |Wolfgang  |Huber     |
CD        |Thomas    |Schreiner |
DVD       |Wolfgang  |Huber     |
SELECT COUNT(*) FROM Persons;
_         |
----------+
5         |
SELECT FirstName,COUNT(LastName) FROM Persons;
FirstName |_         |
----------+----------+
Wolfgang  |5         |
Wolfgang  |5         |
Thomas    |5         |
Thomas    |5         |
Thomas    |5         |
SELECT FirstName,COUNT(LastName) FROM Persons GROUP BY FirstName;
FirstName |_         |
----------+----------+
Wolfgang  |2         |
Thomas    |3         |
SELECT FirstName,COUNT(LastName) FROM Persons GROUP BY FirstName HAVING
  (FirstName = "Wolfgang");
FirstName |_         |
----------+----------+
Wolfgang  |2         |
SELECT * FROM Persons WHERE (FirstName LIKE "%olf%");
Per|LastName  |FirstName |Address             |
---+----------+----------+--------------------+
2  |Schreiner |Wolfgang  |Linz                |
3  |Huber     |Wolfgang  |Linz                |
SELECT * FROM Persons JOIN Sales ON ((Persons.PersonId = Sales.PersonId));
Per|LastName  |FirstName |Address             |Per|Art|
---+----------+----------+--------------------+---+---+
2  |Schreiner |Wolfgang  |Linz                |2  |1  |
2  |Schreiner |Wolfgang  |Linz                |2  |2  |
3  |Huber     |Wolfgang  |Linz                |3  |1  |
3  |Huber     |Wolfgang  |Linz                |3  |2  |
3  |Huber     |Wolfgang  |Linz                |3  |3  |
4  |Schreiner |Thomas    |Linz                |4  |2  |
SELECT * FROM Persons JOIN Sales USING (PersonId);
Per|LastName  |FirstName |Address             |Per|Art|
---+----------+----------+--------------------+---+---+
2  |Schreiner |Wolfgang  |Linz                |2  |1  |
```

```
2  |Schreiner |Wolfgang  |Linz                |2  |2  |
3  |Huber     |Wolfgang  |Linz                |3  |1  |
3  |Huber     |Wolfgang  |Linz                |3  |2  |
3  |Huber     |Wolfgang  |Linz                |3  |3  |
4  |Schreiner |Thomas    |Linz                |4  |2  |
SELECT * FROM Persons JOIN Sales ON ((Persons.PersonId = Sales.PersonId))
  JOIN Articles ON ((Sales.ArticleId = Articles.ArticleId));
Per|LastName  |FirstName |Address             |Per|Art|Art|Name      |Price|
---+----------+----------+--------------------+---+---+---+----------+-----+
2  |Schreiner |Wolfgang  |Linz                |2  |1  |1  |Book      |10   |
2  |Schreiner |Wolfgang  |Linz                |2  |2  |2  |CD        |20   |
3  |Huber     |Wolfgang  |Linz                |3  |1  |1  |Book      |10   |
3  |Huber     |Wolfgang  |Linz                |3  |2  |2  |CD        |20   |
3  |Huber     |Wolfgang  |Linz                |3  |3  |3  |DVD       |30   |
4  |Schreiner |Thomas    |Linz                |4  |2  |2  |CD        |20   |
SELECT * FROM Persons JOIN Sales USING (PersonId) JOIN Articles USING (ArticleId);
Per|LastName  |FirstName |Address             |Per|Art|Art|Name      |Price|
---+----------+----------+--------------------+---+---+---+----------+-----+
2  |Schreiner |Wolfgang  |Linz                |2  |1  |1  |Book      |10   |
2  |Schreiner |Wolfgang  |Linz                |2  |2  |2  |CD        |20   |
3  |Huber     |Wolfgang  |Linz                |3  |1  |1  |Book      |10   |
3  |Huber     |Wolfgang  |Linz                |3  |2  |2  |CD        |20   |
3  |Huber     |Wolfgang  |Linz                |3  |3  |3  |DVD       |30   |
4  |Schreiner |Thomas    |Linz                |4  |2  |2  |CD        |20   |
CREATE TABLE Numbers(Number INT(10));
INSERT INTO Numbers VALUES (1);
INSERT INTO Numbers VALUES (2);
INSERT INTO Numbers VALUES (2);
INSERT INTO Numbers VALUES (3);
SELECT * FROM Numbers;
Number    |
----------+
1         |
2         |
2         |
3         |
SELECT DISTINCT * FROM Numbers;
Number    |
----------+
1         |
2         |
3         |
(SELECT * FROM Numbers) UNION DISTINCT (SELECT * FROM Numbers);
Number    |
----------+
1         |
2         |
3         |
(SELECT * FROM Numbers) UNION ALL (SELECT * FROM Numbers);
Number    |
----------+
1         |
2         |
2         |
```

```
3         |
1         |
2         |
2         |
3         |
(SELECT * FROM Numbers) INTERSECT (SELECT * FROM Numbers);
Number    |
----------+
1         |
2         |
3         |
(SELECT * FROM Numbers) EXCEPT (SELECT * FROM Numbers);
Number    |
----------+
SELECT * FROM Persons,Sales,Articles WHERE (((Persons.PersonId = Sales.PersonId)
  AND (Sales.ArticleId = Articles.ArticleId))
  AND (Persons.PersonId = (SELECT MIN(PersonId) FROM Persons)));
Per|LastName  |FirstName |Address             |Per|Art|Art|Name      |Price|
---+----------+----------+--------------------+---+---+---+----------+-----+
2  |Schreiner |Wolfgang  |Linz                |2  |1  |1  |Book      |10   |
2  |Schreiner |Wolfgang  |Linz                |2  |2  |2  |CD        |20   |
SELECT * FROM Persons,Sales WHERE (Persons.PersonId =
  (SELECT (ArticleId+MIN(PersonId)) FROM Persons));
Per|LastName  |FirstName |Address             |Per|Art|
---+----------+----------+--------------------+---+---+
3  |Huber     |Wolfgang  |Linz                |2  |1  |
3  |Huber     |Wolfgang  |Linz                |3  |1  |
4  |Schreiner |Thomas    |Linz                |2  |2  |
4  |Schreiner |Thomas    |Linz                |3  |2  |
4  |Schreiner |Thomas    |Linz                |4  |2  |
```