The Third Manifesto

(version dated April 5th, 2016, superseding all previous versions)

... the powerful plain third manifesto
—with apologies to Stephen Spender

These principles are eternal, and will remain eternal
—unidentified politician, quoted in a recent news item

This is the fourth revision of the version given in Chapter 1 of *Database Explorations* by C. J. Date and Hugh Darwen, Trafford, 2010. What follows is a revision of the body of that chapter, with altered and new text shown in blue.

Revision history:

Revision 2, February 7th, 2013, affected RM Prescriptions 1 and 8 only.

Revision 3, October 31st, 2013, merely removed the extraneous braces from the symbols {*H*}, {*H1*}, {*H2*}, and {*b*}.

Revision 4, September 18th, 2015, affected RM Prescription 16 only.

Revision 5, April 5th, 2016, added a note to RM Prescription 8 concerning indeterminate operators.

Chapter 1 in *Database Explorations* provides a precise and succinct definition of the various components that go to make up *The Third Manifesto* ("the *Manifesto*" for short). The bulk of the chapter consists of a revised version of Chapter 4 from the book *Databases, Types, and the Relational Model: The Third Manifesto,* 3rd edition, by C. J. Date and Hugh Darwen, Addison-Wesley, 2006 ("the *Manifesto* book" for short). The principal revisions are as follows:

- This introductory section has been added. Its purpose is to make the chapter more self-contained by providing definitions and explanations of terms used in the rest of the chapter.
- Numerous changes have been made at the detail level.
- The final section ("Recent *Manifesto* Changes") has been completely rewritten. In the *Manifesto* book, it served to summarize differences between the *Manifesto* as defined therein and the version defined in that book's predecessor (viz., *Foundation for Future Database Systems: The Third Manifesto,* 2nd edition, by C. J. Date and Hugh Darwen, Addison-Wesley, 2000); now it summarizes differences between the *Manifesto* as defined in the present chapter and the version defined in the *Manifesto* book.

Here now are the promised definitions and explanations of terms (extracted, mostly, from earlier chapters of the *Manifesto* book but reworded somewhat here):

- ***D:*** The *Manifesto* makes repeated reference to a hypothetical language it calls **D**. However, the name **D** is merely a useful generic label; any language that conforms to the principles laid down in the *Manifesto* is a valid **D** (and any language that fails so to conform is not a valid **D**).
- ***Tutorial D:*** The *Manifesto* book includes a fairly formal (though certainly not rigorous) definition of a particular **D** it calls **Tutorial D**. **Tutorial D** is a computationally complete programming language with fully integrated database functionality. However, it's deliberately not meant to be industrial strength;

rather, it's a "toy" language, whose principal purpose is to serve as a teaching vehicle. Thus, many features that would be required in an industrial strength language are intentionally omitted; in particular, it includes no exception handling, no I/O facilities, and no authorization features of any kind.

- *"RM" and "OO":* The *Manifesto* defines a number of prescriptions and proscriptions that **D** is required to adhere to. Prescriptions that arise from the relational model are called *Relational Model Prescriptions* (*RM Prescriptions*). Prescriptions that do not arise from the relational model are called *Other Orthogonal Prescriptions* (*OO Prescriptions*). Proscriptions are similarly divided into RM and OO categories. The *Manifesto* also includes a series of *Very Strong Suggestions,* likewise divided into RM and OO categories.

- *Expression:* The term *expression* refers to the representation in concrete syntactic form of a read-only operator invocation. Observe in particular that variable references are regarded as expressions in exactly this sense; so too are constant references (see RM Prescription 19). *Note:* Two important examples of the latter, not explicitly referenced in the *Manifesto* as such but supported by **Tutorial D,** are TABLE_DUM and TABLE_DEE. TABLE_DEE is the unique relation with no attributes and just one tuple—the empty tuple, of course—and TABLE_DUM is the unique relation with no attributes and no tuples at all.

- *Literal:* A literal is an expression denoting a selector operator invocation (see RM Prescriptions 4, 9, and 10) in which every argument expression is a literal in turn. In other words, a literal is, loosely, what's sometimes called a *self-defining symbol;* i.e., it's a symbol that denotes a value that's fixed and determined by the symbol in question, and hence can be determined at compile time (and the type of that value is therefore also fixed and determined by the symbol in question, and can also be determined at compile time). Observe that there's a logical difference between a literal as such and the value it denotes.

- *Argument and argument expression:* An argument is what's substituted for a parameter when an operator is invoked; it's denoted by an argument expression, which is part of the representation in concrete syntactic form of the operator invocation in question. An argument is either a value or a variable. To be specific, if the parameter in question is subject to update (see RM Prescription 3), the argument must be a variable (and the corresponding argument expression must be a variable reference specifically, denoting the variable in question); otherwise it must be a value (though the corresponding argument expression might still be just a variable reference, denoting in this case the current value of the variable in question).

- *Scalar:* Loosely, a type is *scalar* if and only if it has no user visible components, and *nonscalar* if and only if it's not scalar; and values, variables, attributes, operators, parameters, and expressions of some type *T* are scalar or nonscalar according as type *T* itself is scalar or nonscalar. But these definitions are only informal, and the *Manifesto* doesn't rely on the scalar vs. nonscalar distinction in any formal sense. For the purposes of the *Manifesto,* in fact, the term *scalar type* can be taken to mean a type that's neither a tuple type nor a relation type, and the term *nonscalar type* can be taken to mean a type that is either a tuple type or a relation type. The terms *scalar value, nonscalar value, scalar operator, nonscalar operator,* etc., can be interpreted analogously.

- *Ordered type:* An ordered type is a type for which a total ordering is defined. Thus, if *T* is such a type and *v1* and *v2* are values of type *T,* then (with respect to that ordering) exactly one of the following comparisons returns TRUE and the other two return FALSE:

  ```
  v1 < v2     v1 = v2     v1 > v2
  ```

One last point: The *Manifesto* book also includes a detailed proposal for a model of type inheritance. However, everything to do with that inheritance model is ignored in the *Manifesto* per se, except for very brief mentions in RM Prescription 1, OO Prescription 2, and OO Very Strong Suggestion 1. The concepts of the inheritance model extend, but do not otherwise invalidate, the concepts of the *Manifesto* per se.

RM PRESCRIPTIONS

1. A **scalar data type** (**scalar type** for short) is a named set of scalar values (**scalars** for short). Given an arbitrary pair of distinct scalar types named *T1* and *T2,* respectively, with corresponding sets of scalar values *S1* and *S2,* respectively, the names *T1* and *T2* shall be distinct and the sets *S1* and *S2* shall be disjoint; in other words, two scalar types shall be equal—i.e., the same type—if and only if they have the same name (and therefore the same set of values). **D** shall provide facilities for users to define their own scalar types (*user defined* scalar types); other scalar types shall be provided by the system (*built in* or *system defined* scalar types). With the sole exception of the system defined empty type *omega* (which is defined only if type inheritance is supported—see OO Prescription 2), the definition of any given scalar type *T* shall be accompanied by a specification of an **example value** of that type. **D** shall also provide facilities for users to destroy user defined scalar types. The system defined scalar types shall include type **boolean** (containing just two values, here denoted TRUE and FALSE), and **D** shall support all four monadic and 16 dyadic logical operators, directly or indirectly, for this type.

2. All scalar values shall be **typed**—i.e., such values shall always carry with them, at least conceptually, some identification of the type to which they belong.

3. A **scalar operator** is an operator that, when invoked, returns a scalar value (the **result** of that invocation). **D** shall provide facilities for users to define and destroy their own scalar operators (*user defined* scalar operators). Other scalar operators shall be provided by the system (*built in* or *system defined* scalar operators). Let *Op* be a scalar operator. Then:

   a. *Op* shall be **read-only,** in the sense that invoking it shall cause no variables to be updated other than ones local to the code that implements *Op.*

   b. Every invocation of *Op* shall denote a value ("produce a result") of the same type, the **result type**—also called the **declared type**—of *Op* (as well as of that invocation of *Op* in particular). The definition of *Op* shall include a specification of that declared type.

   c. The definition of *Op* shall include a specification of the type of each parameter to *Op,* the **declared type** of that parameter. If parameter *P* is of declared type *T,* then, in every invocation of *Op,* the expression that denotes the argument corresponding to *P* in that invocation shall also be of type *T,* and the value denoted by that expression shall be **effectively assigned** to *P*. *Note:* The prescriptions of this paragraph c. shall also apply if *Op* is an update operator instead of a read-only operator (see below).

   It is convenient to deal with update operators here as well, despite the fact that such operators are not scalar (nor are they nonscalar—in fact, they are not typed at all). An **update operator** is an operator that, when invoked, is allowed to update at least one variable that is not local to the code that implements that operator. Let *V* be such a variable. If the operator accesses *V* via some parameter *P*, then that parameter *P* is **subject to update**. **D** shall provide facilities for users to define and destroy their own update operators (*user defined* update operators). Other update operators shall be provided by the system (*built in* or *system defined* update operators). Let *Op* be an update operator. Then:

   d. No invocation of *Op* shall denote a value ("produce a result").

   e. The definition of *Op* shall include a specification of which parameters to *Op* are subject to update. If parameter *P* is subject to update, then, in every invocation of *Op,* the expression that denotes the argument corresponding to *P* in that invocation shall be a variable reference specifically, and, on completion of the execution of *Op* caused by that invocation, the final value assigned to *P* during that execution shall be **effectively assigned** to that variable.

4. Let *T* be a scalar type, and let *v* be an appearance in some context of some value of type *T*. By definition, *v* has exactly one **physical representation** and one or more **possible representations** (at least one, because there is obviously always one that is the same as the physical representation). Physical representations for values of type *T* shall be specified by means of some kind of *storage structure definition language* and shall not be visible in **D**. As for possible representations:

   a. If *T* is user defined, then at least one possible representation for values of type *T* shall be declared and thus made visible in **D**. For each possible representation *PR* for values of type *T* that is visible in **D,** exactly one **selector** operator *S*, of declared type *T*, shall be provided. That operator *S* shall have all of the following properties:

      1. There shall be a one to one correspondence between the parameters of *S* and the components of *PR* (see RM Prescription 5). Each parameter of *S* shall have the same declared type as the corresponding component of *PR*.

      2. Every value of type *T* shall be produced by some invocation of *S* in which every argument expression is a literal.

      3. Every successful invocation of *S* shall produce some value of type *T*.

   b. If *T* is system defined, then zero or more possible representations for values of type *T* shall be declared and thus made visible in **D**. A possible representation *PR* for values of type *T* that is visible in **D** shall behave in all respects as if *T* were user defined and *PR* were a declared possible representation for values of type *T*. If no possible representation for values of type *T* is visible in **D,** then at least one **selector** operator *S*, of declared type *T*, shall be provided. Each such selector operator shall have all of the following properties:

      1. Every argument expression in every invocation of *S* shall be a literal.

      2. Every value of type *T* shall be produced by some invocation of *S*.

      3. Every successful invocation of *S* shall produce some value of type *T*.

5. Let some declared possible representation *PR* for values of scalar type *T* be defined in terms of components *C1, C2, ..., Cn* ($n \geqslant 0$), each of which has a name and a declared type. Let *v* be a value of type *T*, and let *PR*(*v*) denote the possible representation corresponding to *PR* for that value *v*. Then *PR*(*v*) shall be **exposed**—i.e., a set of read-only and update operators shall be provided such that:

   a. For all such values *v* and for all *i* ($i = 1, 2, ..., n$), it shall be possible to "retrieve" (i.e., read the value of) the *Ci* component of *PR*(*v*). The read-only operator that provides this functionality shall have declared type the same as that of *Ci*.

   b. For all variables *V* of declared type *T* and for all *i* ($i = 1, 2, ..., n$), it shall be possible to update *V* in such a way that if the values of *V* before and after the update are *v* and *v'* respectively, then the possible representations corresponding to *PR* for *v* and *v'* (i.e., *PR*(*v*) and *PR*(*v'*), respectively) differ in their *Ci* components.

   Such a set of operators shall be provided for each possible representation declared for values of type *T*.

6. **D** shall support the **TUPLE** type generator. That is, given some heading *H* (see RM Prescription 9), **D** shall support use of the **generated type** TUPLE *H* as a basis for defining (or, in the case of values, selecting):

   a. Values of that type (see RM Prescription 9)

b. Variables of that type (see RM Prescription 12)

c. Attributes of that type (see RM Prescriptions 9 and 10)

d. Components of that type within declared possible representations (see RM Prescription 5)

e. Read-only operators of that type (see RM Prescription 20)

f. Parameters of that type to user defined operators (see RM Prescriptions 3 and 20)

The generated type TUPLE *H* shall be referred to as a **tuple type,** and the name of that type shall be, precisely, TUPLE *H*. The terminology of **degree, attributes,** and **heading** introduced in RM Prescription 9 shall apply, mutatis mutandis, to that type, as well as to values and variables of that type (see RM Prescription 12). Tuple types TUPLE *H1* and TUPLE *H2* shall be equal if and only if *H1* = *H2*. The applicable operators shall include operators analogous to the RENAME, *project,* EXTEND, and JOIN operators of the relational algebra (see RM Prescription 18), together with tuple assignment (see RM Prescription 21) and tuple comparisons (see RM Prescription 22); they shall also include (a) a tuple selector operator (see RM Prescription 9), (b) an operator for extracting a specified attribute value from a specified tuple (the tuple in question might be required to be of degree one—see RM Prescription 9), and (c) operators for performing tuple "nesting" and "unnesting."

Note: When we say "the name of [a certain tuple type] shall be, precisely, TUPLE *H*," we do not mean to prescribe specific syntax. The *Manifesto* does not prescribe syntax. Rather, what we mean is that the type in question shall have a name that does both of the following, no more and no less: First, it shall specify that the type is indeed a tuple type; second, it shall specify the pertinent heading. Syntax of the form "TUPLE *H*" satisfies these requirements, and we therefore use it as a convenient shorthand; however, all appearances of that syntax throughout this *Manifesto* are to be interpreted in the light of these remarks.

7. **D** shall support the **RELATION** type generator. That is, given some heading *H* (see RM Prescription 9), **D** shall support use of the **generated type** RELATION *H* as the basis for defining (or, in the case of values, selecting):

a. Values of that type (see RM Prescription 10)

b. Variables of that type (see RM Prescription 13)

c. Attributes of that type (see RM Prescriptions 9 and 10)

d. Components of that type within declared possible representations (see RM Prescription 5)

e. Read-only operators of that type (see RM Prescription 20)

f. Parameters of that type to user defined operators (see RM Prescriptions 3 and 20)

The generated type RELATION *H* shall be referred to as a **relation type,** and the name of that type shall be, precisely, RELATION *H*. The terminology of **degree, attributes,** and **heading** introduced in RM Prescription 9 shall apply, mutatis mutandis, to that type, as well as to values and variables of that type (see RM Prescription 13). Relation types RELATION *H1* and RELATION *H2* shall be equal if and only if *H1* = *H2*. The applicable operators shall include the usual operators of the relational algebra (see RM Prescription 18), together with relational assignment (see RM Prescription 21) and relational comparisons (see RM Prescription 22); they shall also include (a) a relation selector operator (see RM Prescription 10), (b) an operator for extracting the sole tuple from a specified relation of cardinality one (see RM Prescription 10), and (c) operators for performing relational "nesting" and "unnesting."

Note: When we say "the name of [a certain relation type] shall be, precisely, RELATION *H*," we do not mean to prescribe specific syntax. The *Manifesto* does not prescribe syntax. Rather, what we mean is that the type in question shall have a name that does both of the following, no more and no less: First, it shall specify that the type is indeed a relation type; second, it shall specify the pertinent heading. Syntax of the form “RELATION *H*” satisfies these requirements, and we therefore use it as a convenient shorthand; however, all appearances of that syntax throughout this *Manifesto* are to be interpreted in the light of these remarks.

8. **D** shall support the **equality** comparison operator “=” for every type *T*. Let *v1* and *v2* be values, and consider the equality comparison *v1* = *v2*. The values *v1* and *v2* shall be of the same type *T*. The comparison shall return TRUE if and only if *v1* and *v2* are the very same value. *Note:* It follows from this prescription that if (a) there exists an operator *Op* (other than “=” itself) with a parameter *P* of declared type *T* such that (b) two successful invocations of *Op* that are identical in all respects except that the argument corresponding to *P* is *v1* in one invocation and *v2* in the other are distinguishable in their effect, then (c) *v1* = *v2* must evaluate to FALSE.

   Note: By "operator" here we mean one that is a function, i.e., determinate. That two evaluations of RANDOM(5), for example, might differ in their results is irrelevant. Also, an operator whose implementation references a variable that is not defined locally within that implementation is indeterminate.

9. A **heading** *H* is a set of ordered pairs or **attributes** of the form <*A*,*T*>, where:

   a. *A* is the name of an **attribute** of *H*. No two distinct pairs in *H* shall have the same attribute name.

   b. *T* is the name of the **declared type** of attribute *A* of *H*.

   The number of pairs in *H*—equivalently, the number of attributes of *H*—is the **degree** of *H*.

   Now let *t* be a set of ordered triples <*A*,*T*,*v*>, obtained from *H* by extending each ordered pair <*A*,*T*> to include an arbitrary value *v* of type *T*, called the **attribute value** for attribute *A* of *t*. Then *t* is a **tuple value** (**tuple** for short) that **conforms** to heading *H*; equivalently, *t* is of the corresponding tuple type (see RM Prescription 6). The degree of that heading *H* shall be the **degree** of *t*, and the attributes and corresponding types of that heading *H* shall be the **attributes** and corresponding **declared attribute types** of *t*.

   Given a heading *H*, exactly one **selector** operator *S*, of declared type TUPLE *H*, shall be provided for selecting an arbitrary tuple conforming to *H*. That operator *S* shall have all of the following properties:

   1. There shall be a one to one correspondence between the parameters of *S* and the attributes of *H*. Each parameter of *S* shall have the same declared type as the corresponding attribute of *H*.

   2. Every tuple of type TUPLE *H* shall be produced by some invocation of *S* in which every argument expression is a literal.

   3. Every successful invocation of *S* shall produce some tuple of type TUPLE *H*.

10. A **relation value** *r* (**relation** for short) consists of a *heading* and a *body*, where:

    a. The **heading** of *r* shall be a heading *H* as defined in RM Prescription 9; *r* **conforms** to that heading; equivalently, *r* is of the corresponding relation type (see RM Prescription 7). The degree of that heading *H* shall be the **degree** of *r*, and the attributes and corresponding types of that heading *H* shall be the **attributes** and corresponding **declared attribute types** of *r*.

    b. The **body** of *r* shall be a set *b* of tuples, all having that same heading *H*. The cardinality of that body shall be the **cardinality** of *r*. *Note:* Relation *r* is an *empty relation* if and only if the set *b* is empty.

    Given a heading *H*, exactly one **selector** operator *S*, of declared type RELATION *H*, shall be provided for selecting an arbitrary relation conforming to *H*. That operator *S* shall have all of the following properties:

1. The sole argument to any given invocation of *S* shall be a set *b* of tuples, each of which shall be denoted by a tuple expression of declared type TUPLE *H*.

2. Every relation of type RELATION *H* shall be produced by some invocation of *S* for which the tuple expressions that together denote the argument to that invocation are all literals.

3. Every successful invocation of *S* shall produce some relation of type RELATION *H*: to be specific, the relation of type RELATION *H* with body *b*.

11. **D** shall provide facilities for users to define **scalar variables**. Each scalar variable shall be named and shall have a specified (scalar) **declared type**. Let scalar variable *V* be of declared type *T;* for so long as variable *V* exists, it shall have a value that is of type *T*. Defining *V* shall have the effect of initializing *V* to some value—either a value specified explicitly as part of the operation that defines *V*, or some implementation defined value otherwise. *Note:* Omitting an explicit initialization value does not preclude the implementation from checking that no reference is made to scalar variable *V* until an explicit assignment to *V* has occurred. Analogous remarks apply to tuple variables (see RM Prescription 12), real relvars (see RM Prescription 14), and private relvars (again, see RM Prescription 14).

12. **D** shall provide facilities for users to define **tuple variables**. Each tuple variable shall be named and shall have a specified **declared type** of the form TUPLE *H* for some heading *H*. Let variable *V* be of declared type TUPLE *H*; then the degree of that heading *H* shall be the **degree** of *V*, and the attributes and corresponding types of that heading *H* shall be the **attributes** and corresponding **declared attribute types** of *V*. For so long as variable *V* exists, it shall have a value that is of type TUPLE *H*. Defining *V* shall have the effect of initializing *V* to some value—either a value specified explicitly as part of the operation that defines *V*, or some implementation defined value otherwise.

13. **D** shall provide facilities for users to define **relation variables** (**relvars** for short)—both database relvars (i.e., relvars that are part of some database) and application relvars (i.e., relvars that are local to some application). **D** shall also provide facilities for users to destroy database relvars. Each relvar shall be named and shall have a specified **declared type** of the form RELATION *H* for some heading *H*. Let variable *V* be of declared type RELATION *H*; then the degree of that heading *H* shall be the **degree** of *V*, and the attributes and corresponding types of that heading *H* shall be the **attributes** and corresponding **declared attribute types** of *V*. For so long as variable *V* exists, it shall have a value that is of type RELATION *H*.

14. Database relvars shall be either *real* or *virtual*. A **virtual relvar** *V* shall be a database relvar whose value at any given time is the result of evaluating a certain relational expression at that time; the relational expression in question shall be specified when *V* is defined and shall mention at least one database relvar other than *V*. A **real relvar** (also known as a **base relvar**) shall be a database relvar that is not virtual. Defining a real relvar *V* shall have the effect of initializing *V* to some value—either a value specified explicitly as part of the operation that defines *V*, or the empty relation of type RELATION *H* otherwise (where RELATION *H* is the type of relvar *V*).

    Application relvars shall be either *public* or *private*. A **public relvar** shall be an application relvar that constitutes the perception on the part of the application in question of some portion of some database. A **private relvar** shall be an application relvar that is completely private to the application in question and is not part of any database. Defining a private relvar *V* shall have the effect of initializing *V* to some value—either a value specified explicitly as part of the operation that defines *V*, or the empty relation of type RELATION *H* otherwise (where RELATION *H* is the type of relvar *V*).

15. Every relvar shall have at least one **candidate key**. At least one such key shall be defined, explicitly or implicitly, at the time the relvar in question is defined, and it shall not be possible to destroy all of the candidate keys of a given relvar other than by destroying the relvar itself.

16. A **database** shall be a named container for relvars; the content of a given database at any given time shall be a set of database relvars. The necessary operators for defining and destroying databases shall have no effect, when invoked, on any current transactions (see RM Prescription 17 and OO Prescriptions 4 and 5), nor on the **D** environment in which any such transaction is operating.

17. Each **transaction** shall interact with exactly one database. However, distinct transactions shall be allowed to interact with distinct databases, and distinct databases shall not necessarily be disjoint. Also, **D** shall provide facilities for a transaction to define new relvars, or destroy existing ones, within its associated database (see RM Prescription 13). Every execution of every **statement** (other than a "begin transaction" statement—see OO Prescription 4) shall be performed within the context of some transaction. Every statement execution shall be **semantically atomic** (i.e., it shall be as if either the statement executes in its entirety or it fails to execute at all), except possibly if either of the following is the case:

    a. The statement in question is not syntactically atomic (i.e., it contains another statement nested inside itself).

    b. The statement in question represents the invocation of a user defined update operator.

18. **D** shall support the usual operators of the **relational algebra** (or some logical equivalent thereof). All such operators shall be expressible without excessive circumlocution. **D** shall support **type inference** for relation types, whereby the type of the result of evaluating an arbitrary relational expression shall be well defined and known to both the system and the user. *Note:* It follows from this prescription that **D** shall also support type inference for tuple types, whereby the type of the result of evaluating an arbitrary tuple expression shall be well defined and known to both the system and the user.

19. **Variable references** and **constant references** shall be valid expressions. The expression *V*, where *V* is a variable reference, shall be regarded as an invocation of a read-only operator that returns the current value of variable *V*. The expression *C*, where *C* is a constant reference, shall be regarded as an invocation of a read-only operator that returns the value of constant *C*.

20. **D** shall provide facilities for users to define and destroy their own **tuple operators** (*user defined* tuple operators) and **relational operators** (*user defined* relational operators), and paragraphs a.-c. from RM Prescription 3 shall apply, mutatis mutandis. **Recursion** shall be permitted in operator definitions.

21. **D** shall support the **assignment** operator ":=" for every type *T*. The assignment shall be referred to as a scalar, tuple, or relation (or relational) assignment according as *T* is a scalar, tuple, or relation type. Let *V* and *v* be a variable and a value, respectively, of the same type. After assignment of *v* to *V* (the "target variable"), the equality comparison *V* = *v* shall evaluate to TRUE (see RM Prescription 8).

    D shall also support a **multiple** form of assignment, in which several individual assignments shall be performed as a single semantically atomic operation. Let *MA* be the multiple assignment

    ```
    A1 , A2 , ... , An ;
    ```

    (where *A1*, *A2*, ..., *An* are individual assignments, each assigning to exactly one target variable, and the semicolon marks the overall end of the operation). Then the semantics of *MA* shall be defined by the following pseudocode (Steps a.-d.):

    a. For *i* := 1 to *n*, expand any syntactic shorthands involved in *Ai*. After all such expansions, let *MA* take the form

```
V1 := X1 , V2 := X2 , ... , Vz := Xz ;
```

for some $z \geqslant n$, where *Vi* is the name of some variable not defined in terms of any others and *Xi* is an expression of declared type the same as that of *Vi*.

b. Let *p* and *q* ($1 \leqslant p < q \leqslant z$) be such that *Vp* and *Vq* are identical and there is no *r* ($r < p$ or $p < r < q$) such that *Vp* and *Vr* are identical. Replace *Vq := Xq* in *MA* by an assignment of the form

```
Vq := WITH ( Vq := Xp ) : Xq
```

and remove *Vp := Xp* from *MA*. Repeat this process until no such pair *p* and *q* remains. Let *MA* now consist of the sequence

```
U1 := Y1 , U2 := Y2 , ... , Um := Ym ;
```

where each *Ui* is some *Vj* ($1 \leqslant i \leqslant j \leqslant m \leqslant z$).

c. For *i* := 1 to *m*, evaluate *Yi*. Let the result be *yi*.

d. For *i* := 1 to *m*, assign *yi* to *Ui*.

Note: Step b. of the foregoing pseudocode makes use of the WITH construct of **Tutorial D**. For further explanation, see Chapter 11 of *Database Explorations* (or Chapter 5 of the *Manifesto* book).

22. **D** shall support certain **comparison operators,** as follows:

a. The operators for comparing scalars shall include “=”, “≠”, and (for ordered types) “<”, “>”, etc.

b. The operators for comparing tuples shall include “=” and “≠” and shall not include “<”, “>”, etc.

c. The operators for comparing relations shall include “=”, “≠”, “⊆” (“is a subset of”), and “⊇” (“is a superset of”) and shall not include “<”, “>”, etc.

d. The operator “∈” for testing membership of a tuple in a relation shall be supported.

In every case mentioned except “∈” the comparands shall be of the same type; in the case of “∈” they shall have the same heading. *Note:* Support for “=” for every type is in fact required by RM Prescription 8.

23. **D** shall provide facilities for defining and destroying **integrity constraints** (**constraints** for short). Let *C* be a constraint. Then *C* can be thought of as a boolean expression (though it might not be explicitly formulated as such); it shall be **satisfied** if and only if that boolean expression evaluates to TRUE, and **violated** if and only if it is not satisfied. No user shall ever see a state of affairs in which *C* is violated. There shall be two kinds of constraints:

a. A **type** constraint shall specify the set of values that constitute a given type.

b. A **database** constraint shall specify that, at all times, values of a given set of database relvars taken in combination shall be such that a given boolean expression (which shall mention no variables other than the database relvars in question) evaluates to TRUE. Insofar as feasible, **D** shall support **constraint inference** for database constraints, whereby the constraints that apply to the result of evaluating an arbitrary relational expression shall be well defined and known to both the system and the user.

Note: Let database relvars *R1* and *R2* be distinct and let database constraint *DBC* mention them both; then assignment of some relation *r1* to *R1* will in general require assignment of some relation *r2* to *R2* in order that *DBC* not be violated. The individual assignments *R1* := *r1* and *R2* := *r2* shall be executed as part of the same multiple assignment operation (see RM Prescription 21). Moreover, if (a) the user requests the assignment *R1* := *r1* without requesting, as part of the same multiple assignment, some assignment to *R2*, but (b) the system is able to determine *r2* (from *DBC* or otherwise) for itself, then (c) the assignment *R2* := *r2* shall be performed automatically (though not necessarily without the user's knowledge) unless (d) such automatic assignments have been declaratively prohibited.

24. Let *DB* be a database; let *DBC1*, *DBC2*, ..., *DBCn* be all of the database constraints defined for *DB* (see RM Prescription 23); and let *DBC* be any boolean expression that is logically equivalent to

```
( DBC1 ) AND ( DBC2 ) AND ... AND ( DBCn ) AND TRUE
```

Then *DBC* shall be the **total database constraint** for *DB*.

25. Every database shall include a set of database relvars that constitute the **catalog** for that database. **D** shall provide facilities for assigning to relvars in the catalog. *Note:* Since assignments in general are allowed to be multiple assignments in particular (see RM Prescription 21), it follows that **D** shall permit any number of operations of a definitional nature—defining and destroying types, operators, variables, constraints, and so on—all to be performed as a single semantically atomic operation.

26. **D** shall be constructed according to well established principles of **good language design**.

RM PROSCRIPTIONS

1. **D** shall include no concept of a "relation" whose attributes are distinguishable by ordinal position. Instead, for every relation *r* expressible in **D,** the attributes of *r* shall be distinguishable by *name*.

2. **D** shall include no concept of a "relation" whose tuples are distinguishable by ordinal position. Instead, for every relation *r* expressible in **D,** the tuples of *r* shall be distinguishable by *value*.

3. **D** shall include no concept of a "relation" containing two distinct tuples *t1* and *t2* such that the comparison "*t1* = *t2*" evaluates to TRUE. It follows that (as already stated in RM Proscription 2), for every relation *r* expressible in **D,** the tuples of *r* shall be distinguishable by value.

4. **D** shall include no concept of a "relation" in which some "tuple" includes some "attribute" that does not have a value.

5. **D** shall not forget that relations with no attributes are respectable and interesting, nor that candidate keys with no components are likewise respectable and interesting.

6. **D** shall include no constructs that relate to, or are logically affected by, the "physical" or "storage" or "internal" levels of the system.

7. **D** shall support no tuple level operations on relvars or relations.

8. **D** shall not include any specific support for "composite" or "compound" attributes, since such functionality can more cleanly be achieved, if desired, through the type support already prescribed.

9. **D** shall include no "domain check override" operators, since such operators are both ad hoc and unnecessary.

10. **D** shall not be called SQL.

OO PRESCRIPTIONS

1. **D** shall permit **compile time type checking.**

2. If **D** supports **type inheritance,** then such support shall conform to the inheritance model defined in Part IV of the *Manifesto* book (as revised in Chapter 19 of *Database Explorations*).

3. **D** shall be **computationally complete**. **D** may support, but shall not require, invocation from "host programs" written in languages other than **D**. **D** may also support, but shall not require, the use of other languages for implementation of user defined operators.

4. **D** shall provide **explicit transaction** support, according to which:

   a. Transaction initiation shall be performed only by means of an explicit **"begin transaction"** statement.

   b. Transaction termination shall be performed only by means of a **"commit"** or **"rollback"** statement; commit must always be explicit, but rollback can be implicit (if and only if the transaction fails through no fault of its own).

   If transaction *TX* terminates with commit ("normal termination"), changes made by *TX* to the applicable database shall be committed. If transaction *TX* terminates with rollback ("abnormal termination"), changes made by *TX* to the applicable database shall be rolled back.

   Optionally, **D** shall also provide **implicit** transaction support, according to which any request to execute some statement *S* (other than a "begin transaction," "commit," or "rollback" statement) while no transaction is in progress shall be treated as if that statement *S* is immediately preceded by a "begin transaction" statement and immediately followed by either a "commit" statement (if statement *S* executes successfully) or a "rollback" statement (otherwise).

5. **D** shall support **nested transactions**—i.e., it shall permit a parent transaction *TX* to initiate a child transaction *TX'* before *TX* itself has terminated, in which case:

   a. *TX* and *TX'* shall interact with the same database (as is in fact required by RM Prescription 17).

   b. Whether *TX* shall be required to suspend execution while *TX'* executes shall be implementation defined. However, *TX* shall not be allowed to terminate before *TX'* terminates; in other words, *TX'* shall be wholly contained within *TX*.

   c. Rollback of *TX* shall include the rolling back of *TX'* even if *TX'* has terminated with commit. In other words, "commit" is always interpreted within the parent context (if such exists) and is subject to override by the parent transaction (again, if such exists).

6. Let *AggOp* be an **aggregate** operator (other than one that simply returns the cardinality of its operand relation), and let the relation over which the aggregation is to be done in some given invocation of *AggOp* be *r*. Without loss of generality, let the items to be aggregated in that invocation of *AggOp* be just the appearances of values within some attribute *A* of *r*. If all of the following are true:

   a. *AggOp* is essentially just shorthand for some iterated dyadic operator *Op* (e.g., the dyadic operator is "+" in the case of SUM)

   b. An identity value *i* exists for *Op* (e.g., the identity value is zero in the case of "+")

   c. The semantics of *AggOp* are not such as to require the result of an invocation to be a value appearing in *A*

   then the invocation is equivalent to *i Op x1 Op x2 ... Op xn*, where *n* ($n \geq 0$) is the cardinality of *r* and *x1, x2, ..., xn* are the *n* appearances of values for *A* in *r*, arbitrarily ordered.

OO PROSCRIPTIONS

1. Relvars are not domains.
2. No database relvar shall include an attribute of type *pointer*.

RM VERY STRONG SUGGESTIONS

1. **D** should provide a mechanism according to which values of some specified candidate key (or specified components thereof) for some specified relvar are **supplied by the system**. It should also provide a mechanism according to which an arbitrary relation can be extended to include an attribute whose values (a) are unique within that relation (or within certain partitions of that relation), and (b) are once again **supplied by the system**.

2. Let *RX* be a relational expression. By definition, *RX* can be thought of as designating a relvar, *R* say—either a user defined relvar (if *RX* is just a relvar name) or a system defined relvar (otherwise). It is desirable, though perhaps not always feasible, for the system to be able to **infer the candidate keys** of *R*, such that:

   a. If *RX* constitutes the defining expression for some virtual relvar *R′*, then those inferred candidate keys can be checked for consistency with the candidate keys explicitly defined for *R′* (if any) and—assuming no conflict—become candidate keys for *R′*.

   b. Those inferred candidate keys can be included in the information about *R* that is made available (in response to a "metaquery") to a user of **D**.

   D should provide such functionality, but without any guarantee (a) that such inferred candidate keys are not proper supersets of actual candidate keys ("proper superkeys") or (b) that such an inferred candidate key is discovered for every actual candidate key.

3. **D** should support **transition constraints**—i.e., constraints on the transitions that a given database can make from one value to another.

4. **D** should provide some shorthand for expressing **quota queries**. It should not be necessary to convert the relation concerned into (e.g.) an array in order to formulate such a query.

5. **D** should provide some shorthand for expressing the **generalized transitive closure** operation, including the ability to specify generalized *concatenate* and *aggregate* operations.

6. **D** should provide some means for users to define their own generic **operators,** including in particular generic **relational** operators.

7. **SQL** should be implementable in **D**—not because such implementation is desirable in itself, but so that a painless migration route might be available for current SQL users. To this same end, existing SQL databases should be convertible to a form that **D** programs can operate on without error.

OO VERY STRONG SUGGESTIONS

1. **Type inheritance** should be supported (in which case, see OO Prescription 2).

2. Let operator *Op* have a parameter of declared type *T*. Then the definition of *Op* should be **logically distinct** from the definition of *T*, not "bundled in" with this latter definition. *Note:* The operators required by RM Prescriptions 4, 5, 8, and 21 might be exceptions in this regard.

3. **D** should support the concept of **single level storage**.

RECENT *MANIFESTO* CHANGES

As indicated in the introduction to this chapter, there are several differences between the *Manifesto* as defined herein and the version defined in Chapter 4 of the *Manifesto* book. For the benefit of readers who might be familiar with that earlier version, we summarize the main differences here. Notes concerning subsequent changes in the present version have been added. Of course, wherever there's a discrepancy, the present version should be taken as superseding.

- RM Prescription 1 has been extended to require that all scalar types, system or user defined, have an associated *example value* (except for the special case of the empty scalar type *omega*, which is part of our inheritance model—see Chapters 19 and 20 of the present book). Note that it's a logical consequence of this new requirement that scalar types are always nonempty (again, except for the special case of type *omega*).[1]

  RM Prescription 1 was further revised in February, 2013. Previously, the first sentence of this Prescription had been "A **scalar data type** (**scalar type** for short) is a named, finite set of scalar values (**scalars** for short)." We have removed the word "finite" because it proved to be controversial. We included that word originally because in practice the set of values that can be supported is constrained by the available memory space, which is finite.

- Several other prescriptions have been revised to drop the explicit requirement that some type be nonempty, since that requirement is now satisfied implicitly.

- Several RM Prescriptions, including RM Prescription 3 in particular, have been reworded slightly to take account of the logical difference between arguments and argument expressions.

- RM Prescription 8 has been simplified, with much of the previous text being recast as a note. In a further revision (February, 2013), the Prescription was extended to require operands of "=" to be of the same type (as had always been intended).

- RM Prescription 10 has been corrected (in fact, the version published in the *Manifesto* book—which was an attempt to clarify the version in the second edition of that book—was seriously in error). It has also been extended slightly to include an explicit definition of the term *empty relation.*

- Bowing to inevitability, RM Prescription 14 has been expanded to allow real relvars to be referred to alternatively as base relvars.

- The second sentence of RM Prescription 16 has been revised. The previous text stated that the necessary operators for defining and destroying databases "shall not be part of **D"**, suggesting that syntax for invoking such commands cannot be included in the language. However, there can be no objection to support in **D** for some "escape" mechanism to support execution of host operating system commands.

- RM Prescription 17 has been extended to make it clear that (a) statements are generally executed within the context of some transaction and (b) such statement executions are always semantically atomic, unless either (a) the statement in question isn't syntactically atomic (e.g., it's a CASE statement or an IF statement), or (b) it's the invocation of some user defined update operator (i.e., it's a CALL statement, or something equivalent to a CALL statement, that causes such an operator to be invoked). Loosely

[1] Thanks to Alfredo Novoa for suggesting the idea of example values. *Note:* We might be persuaded to make specification of such values optional if it can be shown there's a serious requirement for user defined empty types.

speaking, in other words, if we assume statements terminate in semicolons, then the unit of "semantic atomicity" is what comes between consecutive semicolons is (with the sole exception already noted, regarding CALL statements).

- RM Prescription 18 has been extended to include a note pointing out that the requirement that relation type inference be supported applies to tuple types as well, mutatis mutandis. Also, a sentence listing certain relational algebra operators that **D** was required to "support, directly or indirectly" has been deleted, since it added nothing.

- RM Prescription 19 has been completely replaced. Previously it read as follows:

  > **Relvar names** and **relation selector invocations** shall both be valid relational expressions. **Recursion** shall be permitted in relational expressions.

  The part about relvar names has been generalized to cover names of variables (and constants) of all types. The part about relation selector invocations is and always was implicit in other prescriptions (RM Prescription 10 in particular). Finally, the part about recursion was, frankly, always a trifle confused; however, the original intent has been preserved in the revised version of RM Prescription 20.

- The term *ordinal type* (mentioned in RM Prescription 22) has been replaced by the more appropriate term *ordered type*. An ordered type is, as the introduction to this chapter indicates, a type for which a total ordering is defined—implying that if *T* is such a type, then the expression "*v1* < *v2*" is defined for all pairs of values *v1* and *v2* of type *T*, returning TRUE if and only if *v1* precedes *v2* with respect to the applicable ordering. An ordinal type is an ordered type for which certain additional operators are required: *first, last, next, prior,* and possibly others. In **Tutorial D,** for example, type INTEGER is an ordinal type; type RATIONAL, by contrast, is an ordered type but not an ordinal one. (The rationale here is that if *p*/*q* is a rational number, then—in mathematics at least, if not in computer arithmetic—there is no rational number that can be said to be the "next" rational number, immediately following *p*/*q*.)

- RM Prescription 23 has been clarified. It has also been extended slightly to include an explicit definition of what it means for a constraint to be violated. A note has been added to recognize the possibility that assignment to relvar *R1* might necessitate implicit assignment to other relvars as a means of enforcing constraints by the mechanism commoly known as *compensatory actions.*

- A note has been added to RM Prescription 25 to point out an important implication that might not be immediately obvious.

- OO Prescription 4 has been extended to allow **D** to support "implicit transactions." *Note:* In practice, we would expect use of this feature to be limited to the use of **D** in an interactive environment.

- RM Very Strong Suggestion 2 ("foreign key shorthand") has been deleted, and RM Very Strong Suggestions 3-8 have been renumbered accordingly. Arguments supporting this change can be found in Chapter 13 of the present book; in a nutshell, however, we feel that (a) foreign keys as usually understood are unnecessarily limited in their applicability, and (b) the usual shorthand formulation is often longer than its longhand equivalent, anyway.

- OO Very Strong Suggestion 2 ("types and operators unbundled") has been made more precise.

In addition to all of the foregoing, many of the prescriptions, proscriptions, and very strong suggestions have been reworded (in some cases extensively). However, those rewordings in themselves are not intended to induce any changes in what's being described.