Tutorial D

(version dated May 23rd, 2013, superseding all previous versions)

I never use a big, big D—

—W. S. Gilbert: *HMS Pinafore*

The version immediately preceding this one is Chapter 11 in *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.

On **October 30th, 2011,** the Boolean operator EQUIV was added (see the section SCALAR DEFINITIONS and, in SCALAR OPERATIONS, *<agg op inv>* and *<summary spec>*).

On **May 23rd, 2013,** the parentheses surrounding *<grouping>, <ungrouping>, <wrapping>,* and *<unwrapping>* were removed in the productions for *<group>, <ungroup>, <wrap>*, *<unwrap>, <tuple wrap>* and*<tuple unwrap>*.

Chapter 11 of *Database Explorations* consists of a heavily edited version of Chapter 5 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—reference [7] in the list of references near the end of the chapter). Its aim is to provide a reference description for the language **Tutorial D**. **Tutorial D** is computationally complete and includes fully integrated ("native") database functionality, but it's not meant to be industrial strength; rather, it's meant to be a "toy" language, whose principal purpose is to serve as a teaching vehicle. As a consequence, many features that would be needed in an industrial strength language have deliberately been omitted.[1] In particular, I/O support and exception handling are both omitted; so too is all support for type inheritance, though possible extensions to provide this latter support are described in Chapter 21 of the present book [10].

In addition to the foregoing, many minor details, both syntactic and semantic, that would require precise specification in an industrial strength language are also ignored. For example, details of the following are all omitted:

- Language characters, identifiers, scope of names, etc.
- Reserved words (if any), comments,[2] delimiters and separators, etc.
- Operator precedence rules (except for a couple of important special cases)
- "Obvious" syntax rules (e.g., distinct parameters to the same operator must have distinct names)

Despite such omissions, the language is meant to be well designed as far as it goes. Indeed, it must be—for otherwise it wouldn't be a valid **D,** since it would violate RM Prescription 26 (which, as explained in Chapter 1 of the present book, requires every **D** to be constructed according to principles of good language design).

As already noted, **Tutorial D** is computationally complete, implying that entire applications can be written in the language; it isn't just a data sublanguage that relies on some host language to provide the necessary computational capabilities. Also, like most languages currently in widespread use, it's imperative in style—

---

[1] At this point the original chapter in reference [7] added that "extending the language to incorporate such features, thereby turning it into what might be called **Industrial D,** could be a worthwhile project." Reference [9] includes some proposals for such extensions.

[2] In examples elsewhere in this book we show comments as text strings bracketed by "/*" and "*/" delimiters.

though it's worth mentioning that the "data retrieval" portions of the language, being based as they are on relational algebra, could in fact be regarded as a functional language if considered in isolation. *Note:* In practice we would hope that those portions would be implemented in an interactive form as well as in the form of a programming language per se; in other words, we endorse the *dual mode principle* as described in, e.g., reference [2].

As already indicated, **Tutorial D** is a relational language; in some respects, however, it can be regarded as an object language as well. For one thing, it follows RM Very Strong Suggestion 3 in supporting the concept of single level storage (see Chapter 1 of the present book). More important, it supports what is probably the most fundamental feature of object languages: namely, it allows users to define their own types. And since there's no reliance on a host language, there's no "impedance mismatch" between the types available inside the database and those available outside (i.e., there's no need to map between the arbitrarily complex types used in the database and the probably rather simple types provided by some conventional host language). In other words, we agree with the object community's complaint that there's a serious problem in trying to build an interface between a DBMS that allows user defined types and a programming language that doesn't. For example, if the database contains a value of type POLYGON, then in **Tutorial D** that value can be assigned to a local variable also of type POLYGON—there's no need to break it down into, say, a sequence of number pairs representing the *x* and *y* coordinates of the vertices of the polygon in question. Altogether, then, it seems fair to characterize **Tutorial D** as a true "object/relational" language (inasmuch as that term has any objective meaning, that is!—see the *Manifesto* book for further explanation).

Tutorial D has been designed to support all of the prescriptions and proscriptions (though not all of the very strong suggestions) defined in Chapter 1 of the present book. However, it isn't meant to be minimal in any sense—it includes numerous features that are really just shorthand for certain combinations of others. (This remark applies especially to its relational support.) However, the shorthands in question are all specifically designed to be shorthands;[3] in other words, the redundancies are deliberate, and are included for usability reasons.

The bulk of the rest of the chapter consists of a BNF grammar for **Tutorial D**. The grammar is defined by means of essentially standard BNF notation, except for a couple of simplifying extensions that we now explain. Let <*xyz*> denote an arbitrary syntactic category (i.e., anything that appears, or potentially could appear, on the left side of some production rule). Then:

- The syntactic category <*xyz list*> denotes a sequence of zero or more <*xyz*>s in which adjacent <*xyz*>s are separated by one or more "white space" characters.

- The syntactic category <*xyz commalist*> denotes a sequence of zero or more <*xyz*>s in which adjacent <*xyz*>s are separated by a comma (as well as, optionally, one or more "white space" characters before the comma or after it or both).

Observe in particular that most of the various lists and commalists described in what follows are allowed to be empty. The effect of specifying an empty list or commalist is usually obvious; for example, an <*assignment*> for which the immediately contained commalist of <*assign*>s is empty degenerates to a <*no op*> ("no operation"). Occasionally, however, there's something a little more interesting to be said about such cases (see Appendix B at the end of the chapter).

Finally, a few miscellaneous points:

- All syntactic categories of the form <... *name*> are defined to be <*identifier*>s, barring explicit production

---

[3] Note the contrast with SQL here. SQL includes many features that are almost but not quite equivalent—"almost but not quite," because the features in question were designed independently of one another. For example, an expression that involves a GROUP BY can usually but not always be replaced by one that doesn't (and, of course, it's that "but not always" that causes the problems).

rules to the contrary. The category <*identifier*> in turn is terminal and is defined no further here.

- A few of the production rules include an alternative on the right side that consists of an ellipsis "…" followed by plain text. In such cases, the plain text is intended as an informal natural language explanation of the syntactic category being defined (or, more usually, one particular form of that syntactic category).

- Some of the production rules are accompanied by a prose explanation of certain additional syntax rules or the corresponding semantics or both, but only where such explanations seem necessary.

- Braces "{" and "}" in the grammar stand for themselves; i.e., they're symbols in the language being defined, not (as they usually are) symbols of the metalanguage. To be specific, we use braces to enclose commalists of items when the commalist in question is intended to denote a set of some kind, in which case (a) the order in which the items appear within that commalist is immaterial and (b) if an item appears more than once, it's treated as if it appeared just once. In particular, we use braces to enclose the commalist of argument expressions in certain *n*-adic operator invocations (e.g., JOIN, UNION). *Note:* In such cases, if the operator in question is idempotent,[4] then the argument expression commalist truly does represent a set of arguments, and the foregoing remarks apply 100 percent. If the operator is not idempotent, however, then the argument expression commalist represents a bag of arguments, not a set—in which case the order in which the argument expressions appear is still immaterial, but repetition has significance (e.g., the expression SUM{1,2,2} returns 5, not 3). Of the *n*-adic operators defined in what follows, the following (only) are not idempotent: COUNT, SUM, AVG, EQUIV, XOR, EXACTLY, COMPOSE, and XUNION.

- The grammar reflects the logical difference between expressions and statements. An expression denotes a value; it can be thought of as a rule for computing or determining the value in question. A statement doesn't denote a value; instead, it causes some action to occur, such as assigning a value to some variable or changing the flow of control.

- Following on from the previous point: Expressions in general are of two kinds, open and closed. A closed expression is one that isn't open, and it can appear wherever expressions in general are allowed. By contrast, an open expression is one that can appear only in certain contexts, because it contains certain references (either attribute references or possrep component references) whose meaning depends on the context in question. To be precise, an open expression can appear (a) as the <*bool exp*> following the keyword WHERE in a <*where*>, a <*relation delete*>, or a <*relation update*>; (b) as the <*bool exp*> following the keyword CONSTRAINT in a <*possrep constraint def*>; (c) as the expression denoting the source in a <*possrep component assign*> within a <*scalar update*>; (d) as the expression denoting the source in an <*attribute assign*> within a <*tuple extend*>, an <*extend*>, a <*tuple update*>, or a <*relation update*>; (e) as the <*integer exp*> or <*exp*> within an <*agg op inv*> or a <*summary*>; (f) as a subexpression within any of the foregoing; (g) nowhere else. In all cases, the meaning of the "certain references" (within the open expression in question) is explained under the pertinent production rule(s).

- In line with discussions in Chapter 3 of the present book, we introduce REL and TUP as synonyms for the keywords RELATION and TUPLE, respectively. We also introduce DEE and DUM as abbreviations for TABLE_DEE and TABLE_DUM, respectively. Throughout the language, in other words, the keywords RELATION and REL can be used interchangeably, and the same is true for the keywords TUPLE and

---

[4] The dyadic operator *Op* is idempotent if and only if *x Op x* = *x* for all *x*. For example, in logic, inclusive OR is idempotent, but exclusive OR (XOR) is not.

TUP, TABLE_DEE and DEE, and TABLE_DUM and DUM. However, these facts aren't explicitly reflected in the grammar that follows.

- The grammar deliberately includes no explicit production rules for the following syntactic categories:

  <possrep component assign>

  <attribute assign>

  However, a detailed explanation of the former can be found in the discussion of the production rule for <*scalar update*>, and a detailed explanation of the latter can be found in the discussion of the production rules for <*tuple update*>, <*relation update*>, <*tuple extend*>, <*extend*>, and <*summarize*>.

- Finally, except for certain changes at the detail level, the version of **Tutorial D** defined by this grammar provides essentially the same functionality as that defined in Chapter 5 of the *Manifesto* book [7]. In particular, major changes proposed elsewhere in the present book are excluded. However, the text does include pointers to chapters in the present book where such changes are proposed, where appropriate.

In addition to the changes noted above, minor corrections and cosmetic improvements have been made to almost all of the production rules and prose explanations. Of course, wherever there's a discrepancy between the present chapter and Chapter 5 of the *Manifesto* book, the present chapter should be taken as superseding.

COMMON CONSTRUCTS

```
<type spec>
    ::=   <scalar type spec>
        | <nonscalar type spec>

<scalar type spec>
    ::=   <scalar type name>
        | SAME_TYPE_AS ( <scalar exp> )

<nonscalar type spec>
    ::=   <tuple type spec>
        | <relation type spec>

<tuple type spec>
    ::=   <tuple type name>
        | SAME_TYPE_AS ( <tuple exp> )
        | TUPLE SAME_HEADING_AS ( <nonscalar exp> )

<relation type spec>
    ::=   <relation type name>
        | SAME_TYPE_AS ( <relation exp> )
        | RELATION SAME_HEADING_AS ( <nonscalar exp> )

<user op def>
    ::=   <user update op def>
        | <user read-only op def>
```

```
<user update op def>
    ::=   OPERATOR <user op name> ( <parameter def commalist> )
             UPDATES { [ ALL BUT ] <parameter name commalist> } ;
                <statement>
          END OPERATOR
```

The <*parameter def commalist*> is enclosed in parentheses instead of braces, as is the corresponding <*argument exp commalist*> in an invocation of the operator in question (see <*user op inv*>, later), because we follow convention in relying on ordinal position for argument/parameter matching.[5] UPDATES (<*parameter name commalist*>), if specified, identifies parameters that are subject to update; alternatively, UPDATES (ALL BUT <*parameter name commalist*>), if specified, identifies parameters that aren't subject to update. *Note:* The specification ALL BUT has an analogous interpretation in all contexts in which it appears; however, the precise nature of that "analogous interpretation" in other contexts is left as an exercise for the reader.

In practice, it might be desirable to support an external form of <*user update op def*> as well. Syntactically, such a <*user update op def*> would include, not a <*statement*> as above, but rather a reference to an external file that contains the code that implements the operator (possibly written in some different language). It might also be desirable to support a form of <*user update op def*> that includes neither a <*statement*> nor such an external reference; such a <*user update op def*> would define merely what is called a *specification signature* for the operator in question, and the implementation code would then have to be defined elsewhere. Splitting operator definitions into separate pieces in this way is likely to prove particularly useful if type inheritance is supported (see Chapter 21). *Note:* Everything in this paragraph applies to <*user read-only op def*>s as well, mutatis mutandis.

```
<parameter def>
    ::=   <parameter name> <type spec>

<user read-only op def>
    ::=   OPERATOR <user op name> ( <parameter def commalist> )
             RETURNS <type spec> ;
                <statement>
          END OPERATOR
```

The <*user op name*> denotes a scalar, tuple, or relational operator, depending on the <*type spec*> in the RETURNS specification.

```
<user op inv>
    ::=   <user op name> ( <argument exp commalist> )
```

The *i*th entry in the <*argument exp commalist*> corresponds to the *i*th entry in the <*parameter def commalist*> in the <*user op def*> identified by <*user op name*>.

```
<argument exp>
    ::=   <exp>

<exp>
    ::=   <scalar exp>
        | <nonscalar exp>
```

---

[5] This remark is true of read-only as well as update operators. In particular, it's true of scalar selector operators—i.e., the arguments to a <*scalar selector inv*> (see later) are specified in parentheses, even though the corresponding parameters are specified in braces (see <*possrep def*>). See Appendix A ("A Remark on Syntax") at the end of the chapter.

```
<scalar exp>
    ::=   <scalar with exp>
        | <scalar nonwith exp>
```

For further details of <*scalar nonwith exp*>s, see the section "Scalar Operations," later.

```
<scalar with exp>
    ::=   WITH ( <name intro commalist> ) : <scalar exp>
```

Let *SWE* be a <*scalar with exp*>, and let *NIC* and *SE* be the <*name intro commalist*> and the <*scalar exp*>, respectively, immediately contained in *SWE*. The individual <*name intro*>s in *NIC* are evaluated in sequence as written. As the next production rule shows, each such <*name intro*> immediately contains an <*introduced name*> and an <*exp*>. Let *NI* be one of those <*name intro*>s, and let the <*introduced name*> and the <*exp*> immediately contained in *NI* be *N* and *X,* respectively. Then *N* denotes the value obtained by evaluating *X,* and it can appear subsequently in *SWE* wherever the expression (*X*)—i.e., *X* in parentheses—would be allowed. *Note:* Everything in this paragraph applies to <*tuple with exp*>s and <*relation with exp*>s as well, mutatis mutandis.

```
<name intro>
    ::=   <introduced name> := <exp>

<nonscalar exp>
    ::=   <tuple exp>
        | <relation exp>

<tuple exp>
    ::=   <tuple with exp>
        | <tuple nonwith exp>
```

For further details of <*tuple nonwith exp*>s, see the section "Tuple Operations," later.

```
<tuple with exp>
    ::=   WITH ( <name intro commalist> ) : <tuple exp>

<relation exp>
    ::=   <relation with exp>
        | <relation nonwith exp>
```

For further details of <*relation nonwith exp*>s, see the section "Relational Operations," later.

```
<relation with exp>
    ::=   WITH ( <name intro commalist> ) : <relation exp>

<user op drop>
    ::=   DROP OPERATOR <user op name>

<selector inv>
    ::=   <scalar selector inv>
        | <nonscalar selector inv>

<nonscalar selector inv>
    ::=   <tuple selector inv>
        | <relation selector inv>
```

```
<var ref>
    ::=   <scalar var ref>
        | <nonscalar var ref>

<scalar var ref>
    ::=   <scalar var name>

<nonscalar var ref>
    ::=   <tuple var ref>
        | <relation var ref>

<tuple var ref>
    ::=   <tuple var name>

<relation var ref>
    ::=   <relation var name>

<attribute ref>
    ::=   <attribute name>

<possrep component ref>
    ::=   <possrep component name>

<assignment>
    ::=   <assign commalist>

<assign>
    ::=   <scalar assign>
        | <nonscalar assign>

<nonscalar assign>
    ::=   <tuple assign>
        | <relation assign>
```

SCALAR DEFINITIONS

```
<scalar type name>
    ::=   <user scalar type name>
        | <built in scalar type name>

<built in scalar type name>
    ::=   INTEGER | RATIONAL | CHARACTER | BOOLEAN
```

As indicated, **Tutorial D** supports the following built in (system defined) scalar types:

- INTEGER (signed integers): synonym INT; literals expressed as an optionally signed decimal integer; usual arithmetic and comparison operators, with usual notation. The (implicit) example value is 0.
- RATIONAL (signed rational numbers): synonym RAT; literals expressed as an optionally signed decimal mantissa (including a decimal point), optionally followed by the letter E and an optionally signed decimal integer exponent (examples: 6., 8.0, 17.5, –4.3E+2); usual arithmetic and comparison operators, with usual notation. The (implicit) example value is 0.0.
- CHARACTER (varying length character strings): synonym CHAR; literals expressed as a sequence,

enclosed in single quotes, of zero or more characters; usual string manipulation and comparison operators, with usual notation—"||" (concatenate), SUBSTR (substring), etc. The (implicit) example value is '' (the empty string). By the way, if you're familiar with SQL, don't be misled here; the SQL data type CHAR corresponds to *fixed* length character strings (the varying length analog is called VARCHAR), and an associated length—default one—must be specified as in, e.g., CHAR(25).[6] **Tutorial D** doesn't support a fixed length character string type.

- BOOLEAN (truth values): synonym BOOL; literals TRUE and FALSE; usual comparison operators ("=" and "≠") and boolean operators (AND, OR, NOT, etc.), with usual notation. The (implicit) example value is FALSE. Note that **Tutorial D**'s support for type BOOLEAN goes beyond that found in many languages in at least three ways:

  1. It includes explicit support for the operators EQUIV and XOR (equivalence and exclusive OR, respectively). The expression *a* EQUIV *b* (where *a* and *b* are <*bool exp*>s) is semantically equivalent to the expression *a* = *b*. The expression *a* XOR *b* (where *a* and *b* are <*bool exp*>s) is semantically equivalent to the expression *a* ≠ *b*.

  2. It supports *n*-adic versions of the operators AND, OR, EQUIV, and XOR. The syntax is:

     ```
     <n-adic bool op name> { <bool exp commalist> }
     ```

     The <*n-adic bool op name*> is AND, OR, EQUIV, or XOR. AND returns TRUE if and only if all specified <*bool exp*>s evaluate to TRUE. OR returns FALSE if and only if all specified <*bool exp*>s evaluate to FALSE. EQUIV returns TRUE if and only if the number of specified <*bool exp*>s that evaluate to FALSE is even. XOR returns TRUE if and only if the number of specified <*bool exp*>s that evaluate to TRUE is odd.

  3. It supports an *n*-adic operator of the form

     ```
     EXACTLY ( <integer exp> , { <bool exp commalist> } )
     ```

     Let the <*integer exp*> evaluate to *n*.[7] Then the overall expression evaluates to TRUE if and only if the number of specified <*bool exp*>s that evaluate to TRUE is *n*.

Type INTEGER is an ordinal type; types RATIONAL and CHARACTER are ordered but not ordinal types. *Note:* In practice we would expect a variety of other system defined scalar types to be supported in addition to the foregoing: DATE, TIME, perhaps BIT (varying length bit strings), and so forth. We omit such types here as irrelevant to our main purpose.

```
<user scalar type def>
    ::=   <user scalar root type def>
```

The syntactic category <*user scalar root type def*> is introduced merely to pave the way for the inheritance support to be described in Chapter 21 (all types are root types—and leaf types too—in the absence of inheritance support).

---

[6] It would be more accurate to refer to CHAR and VARCHAR in SQL not as types but as type generators, and the associated length specification (as in, e.g., CHAR(25)) as the argument to an invocation of such a generator.

[7] The detailed syntax of <*integer exp*>s is left unspecified here (as is that of <*literal*> also—see <*user scalar root type def*>, later); however, we note that an <*integer exp*> is of course a numeric expression, and hence a <*scalar exp*> also.

```
<user scalar root type def>
    ::=   TYPE <user scalar type name>
              [ <ordering> ] <possrep def list>
                INIT ( <literal> )
```

Let *T* be the scalar type being defined. The purpose of the INIT specification is to introduce the example value required for *T* by RM Prescription 1 (and the declared type of the *<literal>* must therefore be *T*). *Note:* Eyebrows might be raised at our choice of keyword here. But RM Prescription 11 requires scalar variables to be initialized, at the time they're defined, to "either a value specified explicitly as part of the operation that defines [the variable in question] *or some implementation defined value otherwise*" (italics added). In **Tutorial D** in particular, if the variable in question happens to be of a user defined type, we use the applicable example value as the necessary implementation defined value;[8] hence our choice of keyword.

```
<ordering>
    ::=   ORDINAL | ORDERED
```

Let *T* be the scalar type being defined. If *<ordering>* is specified, then:

- Type *T* is an ordered type (i.e., the operator "<" is defined for all pairs of values of the type).
- If and only if ORDINAL is specified, then type *T* is also an ordinal type, in which case certain additional operators must also be defined—*first* and *last* (which return the first and last value, respectively, of type *T* with respect to the applicable ordering), and *next* and *prior* (which, given a particular value of type *T,* return that value's successor and predecessor, respectively, again with respect to the applicable ordering). Further details are beyond the scope of this chapter.

```
<possrep def>
    ::=   POSSREP [ <possrep name> ]
                  { <possrep component def commalist>
                           [ <possrep constraint def> ] }
```

If *<possrep name>* is omitted, a *<possrep name>* equal to the *<user scalar type name>* of the containing *<user scalar root type def>* is assumed by default.

```
<possrep component def>
    ::=   <possrep component name> <type spec>
```

No two distinct *<possrep def>*s within the same *<user scalar type def>* can include a component with the same *<possrep component name>*.

```
<possrep constraint def>
    ::=   CONSTRAINT <bool exp>
```

The *<bool exp>* must not mention any variables, but *<possrep component ref>*s can be used to denote the corresponding components of the applicable possible representation ("possrep") of an arbitrary value of the scalar type in question.

```
<user scalar type drop>
    ::=   DROP TYPE <user scalar type name>
```

---

[8] Indeed, we find it hard to believe for *any* **D** that the necessary implementation defined value would be anything other than the applicable example value.

```
<scalar var def>
    ::=  VAR <scalar var name> <scalar type or init value>

<scalar type or init value>
    ::=  <scalar type spec> | INIT ( <scalar exp> )
       | <scalar type spec> INIT ( <scalar exp> )
```

If <*scalar type spec*> and the INIT specification both appear, the declared type of <*scalar exp*> must be the type specified by <*scalar type spec*>. If <*scalar type spec*> appears, the declared type of the scalar variable is the specified type; otherwise it's the same as that of <*scalar exp*>. If the INIT specification appears, the scalar variable is initialized to the value of <*scalar exp*>; otherwise it's initialized to the example value of the pertinent type.

TUPLE DEFINITIONS

```
<tuple type name>
    ::=  TUPLE <heading>

<heading>
    ::=  { <attribute commalist> }

<attribute>
    ::=  <attribute name> <type spec>

<tuple var def>
    ::=  VAR <tuple var name> <tuple type or init value>

<tuple type or init value>
    ::=  <tuple type spec> | INIT ( <tuple exp> )
       | <tuple type spec> INIT ( <tuple exp> )
```

If <*tuple type spec*> and the INIT specification both appear, the declared type of <*tuple exp*> must be the type specified by <*tuple type spec*>. If <*tuple type spec*> appears, the declared type of the tuple variable is the specified type; otherwise it's the same as that of <*tuple exp*>. If the INIT specification appears, the tuple variable is initialized to the value of <*tuple exp*>; otherwise it's initialized to the tuple with the default initialization value for each of its attributes, where (a) the default initialization value for a scalar attribute is the example value of the pertinent scalar type; (b) the default initialization value for a tuple valued attribute is the tuple with the default initialization values for each of the attributes of the tuple type in question; and (c) the default initialization value for a relation valued attribute is the empty relation of the pertinent type.

RELATIONAL DEFINITIONS

```
<relation type name>
    ::=  RELATION <heading>

<relation var def>
    ::=  <database relation var def>
       | <application relation var def>

<database relation var def>
    ::=  <real relation var def>
       | <virtual relation var def>
```

A <*database relation var def*> defines a database relvar—i.e., a relvar that's part of the database. In particular, therefore, it causes an entry to be made in the catalog. Note, however, that neither databases nor catalogs are explicitly mentioned anywhere in the syntax of **Tutorial D**.

```
<real relation var def>
    ::=   VAR <relation var name> <real or base>
              <relation type or init value> <key def list>
```

An empty <*key def list*> is equivalent to a <*key def list*> of the form KEY {ALL BUT}.

```
<real or base>
    ::=   REAL | BASE

<relation type or init value>
    ::=   <relation type spec> | INIT ( <relation exp> )
        | <relation type spec> INIT ( <relation exp> )
```

An INIT specification can appear only if either REAL (or BASE) or PRIVATE is specified for the relvar in question (see <*application relation var def*>, later, for an explanation of PRIVATE). If <*relation type spec*> and the INIT specification both appear, the declared type of <*relation exp*> must be the type specified by <*relation type spec*>. If <*relation type spec*> appears, the declared type of the relation variable is the specified type; otherwise it's the same as that of <*relation exp*>. If and only if the relvar is either real or private, then (a) if the INIT specification appears, the relvar is initialized to the value of <*relation exp*>; (b) otherwise it's initialized to the empty relation of the appropriate type.

```
<key def>
    ::=   KEY { [ ALL BUT ] <attribute ref commalist> }
```

Tutorial D uses the unqualified keyword KEY to mean a candidate key specifically. It doesn't explicitly support primary keys as such; in fact, it makes no distinction between primary and alternate keys. *Note:* Elsewhere this book proposes introducing explicit syntax for foreign keys in addition to candidate keys. See references [5] and [9].

```
<virtual relation var def>
    ::=   VAR <relation var name> VIRTUAL
              ( <relation exp> ) <key def list>
```

The <*relation exp*> must mention at least one database relvar and no other variables. An empty <*key def list*> is equivalent to a <*key def list*> that contains exactly one <*key def*> for each key that can be inferred by the system from <*relation exp*>.

```
<application relation var def>
    ::=   VAR <relation var name> <private or public>
              <relation type or init value> <key def list>
```

An empty <*key def list*> is equivalent to a <*key def list*> of the form KEY {ALL BUT}.

```
<private or public>
    ::=   PRIVATE | PUBLIC

<relation var drop>
    ::=   DROP VAR <relation var ref>
```

The <*relation var ref*> must denote a database relvar, not an application one.

```
<constraint def>
    ::=   CONSTRAINT <constraint name> <bool exp>
```

A <*constraint def*> defines a database constraint. The <*bool exp*> must not reference any variables other than database relvars. (**Tutorial D** doesn't support constraints that reference any other kinds of variables, though there's no logical reason why it shouldn't.)

```
<constraint drop>
    ::=   DROP CONSTRAINT <constraint name>
```

SCALAR OPERATIONS

```
<scalar nonwith exp>
    ::=   <scalar var ref>
        | <scalar op inv>
        | ( <scalar exp> )

<scalar op inv>
    ::=   <user op inv>
        | <built in scalar op inv>
```

In the <*user op inv*> case, the operator being invoked must be a read-only operator specifically.

```
<built in scalar op inv>
    ::=   <scalar selector inv>
        | <THE_ op inv>
        | <attribute extractor inv>
        | <agg op inv><agg op name>
    ::=   COUNT | SUM | AVG | MAX | MIN
        | AND | OR | EQUIV | XOR | EXACTLY
        | UNION | D_UNION | INTERSECT | XUNION
        | ... plus the usual possibilities
```

It's convenient to get "the usual possibilities" out of the way first. By this term, we mean the usual numeric operators ("+", "*", etc.), character string operators ("||", SUBSTR, etc.), and boolean operators, all of which we've already said are built in (system defined) operators in **Tutorial D**. It follows that numeric expressions, character string expressions, and in particular boolean expressions—i.e., <*bool exp*>s—are all <*scalar exp*>s (and we assume the usual syntax in each case). The following are also <*scalar exp*>s:

- Two special forms of <*bool exp*>, IS_EMPTY (<*relation exp*>), which returns TRUE if and only if the relation denoted by <*relation exp*> is empty, and IS_NOT_EMPTY (<*relation exp*>), which returns TRUE if and only if the relation denoted by <*relation exp*> is nonempty.

- CAST expressions of the form CAST_AS_*T* (<*scalar exp*>), where *T* is a scalar type and <*scalar exp*> denotes a scalar value to be converted ("cast") to that type. *Note:* We use syntax of the form CAST_AS_*T* (...), rather than CAST (... AS *T*), because this latter form raises "type TYPE" issues—e.g., what is the type of operand *T?*—that we prefer to avoid.

- IF ... END IF and CASE ... END CASE expressions of the usual form, viz.:

```
IF <bool exp> THEN <scalar exp> [ ELSE <scalar exp> ] END IF

CASE <when def list> [ ELSE <scalar exp> ] END CASE
```

A <*when def*> in turn takes the form:

```
WHEN <bool exp> THEN <scalar exp>
```

Note: We assume without going into details that tuple and relation analogs of the foregoing scalar IF and CASE expressions are available also.

```
<scalar selector inv>
    ::=   <built in scalar literal>
        | <possrep name> ( <argument exp commalist> )
```

The syntax of <*built in scalar literal*> is explained in the prose following the production rule for <*built in scalar type name*>. In the second format, the *i*th entry in the <*argument exp commalist*> corresponds to the *i*th entry in the <*possrep component def commalist*> in the possrep identified by <*possrep name*>. *Note:* Whether scalar selectors are regarded as system defined or user defined could be a matter of some debate, but the point is unimportant for present purposes. Analogous remarks apply to THE_ operators and attribute extractors also (see the next two production rules).

```
<THE_ op inv>
    ::=   <THE_ op name> ( <scalar exp> )
```

We include this production rule in this section because in practice we expect most <*THE_ op inv*>s to denote scalar values. In fact, however, a <*THE_ op inv*> will be a <*scalar exp*>, a <*tuple exp*>, or a <*relation exp*>, depending on the type of the <*possrep component*> corresponding to <*THE_ op name*>.

```
<attribute extractor inv>
    ::=   <attribute ref> FROM <tuple exp>
```

We include this production rule in this section because in practice we expect most attributes to be scalar. In fact, however, an <*attribute extractor inv*> will be a <*scalar exp*>, a <*tuple exp*>, or a <*relation exp*>, depending on the type of <*attribute ref*>.

```
<agg op inv>
    ::=   <agg op name>
            ( [ <integer exp> , ] <relation exp> [ , <exp> ] )
        | <n-adic count etc>
```

In the first format:

- The <*integer exp*> and following comma must be specified if and only if the <*agg op name*> is EXACTLY. The <*exp*> must be omitted if the <*agg op name*> is COUNT.

- Let the relation denoted by the specified <*relation exp*> be *r*. If the <*agg op name*> is not COUNT, then the <*exp*> can be omitted only if *r* is of degree one, in which case an <*exp*> consisting of an <*attribute ref*> that designates the sole attribute of *r* is specified implicitly. More generally, the <*exp*> is allowed to contain an <*attribute ref*>, *AR* say, wherever a <*selector inv*> would be allowed. During the process of computing the desired aggregation, <*exp*> is effectively evaluated for each tuple of *r* in turn. If the <*attribute name*> of *AR* is that of an attribute of *r,* then (for each such evaluation) *AR* denotes the corresponding attribute value from the corresponding tuple; otherwise the <*agg op inv*> must be contained within some expression in which the meaning of *AR* is defined. In other words, the <*agg op inv*>

  ```
  agg ( rx , x )
  ```

  is equivalent to the following:

```
agg ( EXTEND rx : { Temp := x } , Temp )
```

- For SUM, the declared type of the attribute denoted by <*attribute ref*> must be some type for which the operator "+" is defined; for AVG, it must be one for which the operators "+" and "/" are defined; for MAX and MIN, it must be some ordered type; for AND, OR, EQUIV, XOR, and EXACTLY, it must be type BOOLEAN; for UNION, D_UNION, INTERSECT, and XUNION, it must be some relation type.

Note: We include this production rule in this section because in practice we expect most <*agg op inv*>s to denote scalar values. In fact, however, an <*agg op inv*> will be a <*scalar exp*>, a <*tuple exp*>, or a <*relation exp*> depending on the type of the operator denoted by <*agg op name*>.

```
<agg op name>
    ::=   COUNT | SUM | AVG | MAX | MIN
        | AND | OR | EQUIV | XOR | EXACTLY
        | UNION | D_UNION | INTERSECT | XUNION
```

COUNT returns a result of declared type INTEGER; SUM, AVG, MAX, MIN, UNION, D_UNION, INTERSECT, and XUNION return a result of declared type the same as that of the attribute denoted by the applicable <*attribute ref*>;[9] AND, OR, EQUIV, XOR, and EXACTLY return a result of declared type BOOLEAN. *Note:* **Tutorial D** also includes support for certain conventional operators (as opposed to aggregate operators) that are *n*-adic versions of (a) AND, OR, EQUIV, XOR, and EXACTLY (see the section "Scalar Definitions," earlier) and (b) UNION, D_UNION, INTERSECT, JOIN, TIMES, XUNION, and COMPOSE (see the section "Relational Operations," later).[10] Analogously, it also includes support for *n*-adic versions of COUNT, SUM, AVG, MAX, and MIN (see the production rule immediately following).

```
<n-adic count etc>
    ::=   <agg op name> { <exp commalist> }
```

The <*agg op name*> must be COUNT, SUM, AVG, MAX, or MIN, and the <*exp*>s must all be of the same declared type. For SUM, that type must be one for which the operator "+" is defined; for AVG, it must be one for which the operators "+" and "/" are defined; for MAX and MIN, it must be some ordered type. *Note:* For SUM, MAX, and MIN, the <*agg op name*> can optionally have a suffix of the form _*T,* where *T* is a scalar type name (as in, e.g., SUM_INTEGER or, equivalently, SUM_INT) and every <*exp*> in the <*exp commalist*> is of declared type *T*. Such a suffix must be specified if the <*exp commalist*> is empty.

```
<scalar assign>
    ::=   <scalar target> := <scalar exp>
        | <scalar update>

<scalar target>
    ::=   <scalar var ref>
        | <scalar THE_ pv ref>
```

The abbreviation *pv* stands for *pseudovariable*. The grammar presented in this chapter doesn't say as much explicitly, but the general intent is that a pseudovariable reference should be allowed to appear wherever a

---

[9] It might be preferable in practice to define AVG in such a way that, e.g., taking the average of a collection of integers returns a rational number. We do not do so here merely for reasons of simplicity.

[10] It would be possible to define JOIN, TIMES, and COMPOSE aggregate operators also. However, JOIN would always be equivalent to INTERSECT; TIMES would always raise an error, except in the degenerate special case in which the pertinent relations were of degree zero; and COMPOSE would always be equivalent to INTERSECT followed by a projection on no attributes.

variable reference is allowed to appear (speaking a trifle loosely).

```
<scalar THE_ pv ref>
    ::=   <THE_ pv name> ( <scalar target> )
```

The declared type of the <*possrep component*> corresponding to <*THE_ pv name*> must be some scalar type.

```
<scalar update>
    ::=   UPDATE <scalar target> :
                { <possrep component assign commalist> }
```

Let the <*scalar target*>*, ST* say, be of declared type *T*. Every <*possrep component assign*>, *PCA* say, in the <*possrep component assign commalist*> is syntactically identical to an <*assign*> (i.e., a <*scalar assign*>, a <*tuple assign*>, or a <*relation assign*>, as applicable), except that:

- The target of *PCA* must be a <*possrep component target*>, *PCT* say.
- *PCT* must identify, directly or indirectly,[11] some *Ci* (*i* = 1, 2, ..., *n*), where *C1, C2, ..., Cn* are the components of some possrep *PR* for type *T* (the same possrep *PR* in every case).
- *PCA* is allowed to contain a <*possrep component ref*>, *PCR* say, wherever a <*selector inv*> would be allowed, where *PCR* is some *Ci* (*i* = 1, 2, ..., *n*) and denotes the corresponding possrep component value from *ST*.

Steps a. and b. of the definition given for multiple assignment under RM Prescription 21 (see Chapter 1 of the present book) are applied to the <*possrep component assign commalist*>. The result of that application is a <*possrep component assign commalist*> in which each <*possrep component assign*> is of the form

```
Ci := exp
```

for some *Ci,* and no two distinct <*possrep component assign*>s specify the same target *Ci*. Then the original <*scalar update*> is equivalent to the <*scalar assign*>

```
ST := PR ( X1 , X2 , ... , Xn )
```

(*PR* here is the selector operator corresponding to the possrep with the same name.) The arguments *Xi* are defined as follows:

- If a <*possrep component assign*>, *PCA* say, exists for *Ci,* then let the <*exp*> from *PCA* be *X*. For all *j* (*j* = 1, 2, ..., *n*), replace references in *X* to *Cj* by (THE_*Cj*(*ST*)). The version of *X* that results is *Xi*.
- Otherwise, *Xi* is THE_*Ci*(*ST*).

```
<possrep component target>
    ::=   <possrep component ref>
        | <possrep THE_ pv ref>
```

[11] The phrase *directly or indirectly* appears several times in this chapter in contexts like this one. In the present context, we can explain it as follows: Again, let <*possrep component assign*> *PCA* specify <*possrep component target*> *PCT*. Then *PCA* identifies *Ci* as its target directly if *PCT* is *Ci;* it identifies *Ci* as its target indirectly if *PCT* takes the form of a <*possrep THE_ pv ref*> *PTR*, where the argument at the innermost level of nesting within *PTR* is *Ci*. The meaning of the phrase *directly or indirectly* in other similar contexts is analogous.

```
<possrep THE_ pv ref>
    ::=   <THE_ pv name> ( <possrep component target> )
```

```
<scalar comp>
    ::=   <scalar exp> <scalar comp op> <scalar exp>
```

Scalar comparisons are a special case of the syntactic category <*bool exp*>.

```
<scalar comp op>
    ::=   = | ≠ | < | ⩽ | > | ⩾
```

The operators "=" and "≠" apply to all scalar types; the operators "<", "⩽", ">", and "⩾" apply only to ordered types.

TUPLE OPERATIONS

```
<tuple nonwith exp>
    ::=   <tuple var ref>
        | <tuple op inv>
        | ( <tuple exp> )
```

```
<tuple op inv>
    ::=   <user op inv>
        | <built in tuple op inv>
```

```
<built in tuple op inv>
    ::=   <tuple selector inv>
        | <THE_ op inv>
        | <attribute extractor inv>
        | <tuple extractor inv>
        | <tuple project>
        | <n-adic other built in tuple op inv>
        | <monadic or dyadic other built in tuple op inv>
```

```
<tuple selector inv>
    ::=   TUPLE { <tuple component commalist> }
```

```
<tuple component>
    ::=   <attribute name> <exp>
```

```
<tuple extractor inv>
    ::=   TUPLE FROM <relation exp>
```

The <*relation exp*> must denote a relation of cardinality one.

```
<tuple project>
    ::=   <tuple exp> { [ ALL BUT ] <attribute ref commalist> }
```

The <*tuple exp*> must not be a <*monadic or dyadic other built in tuple op inv*>. *Note:* Although we generally have little to say regarding operator precedence, we find it convenient to assign high precedence to tuple projection in particular. A similar remark applies to relational projection as well (see later).

```
<n-adic other built in tuple op inv>
    ::=   <n-adic tuple union>
```

```
<n-adic tuple union>
    ::=   UNION { <tuple exp commalist> }
```

N-adic tuple INTERSECT, COMPOSE, and XUNION operators could also be defined if desired (though *n*-adic tuple COMPOSE and *n*-adic tuple XUNION operators would be logically equivalent).

```
<monadic or dyadic other built in tuple op inv>
    ::=   <monadic other built in tuple op inv>
        | <dyadic other built in tuple op inv>

<monadic other built in tuple op inv>
    ::=   <tuple rename> | <tuple extend>
        | <tuple wrap> | <tuple unwrap>

<tuple rename>
    ::=   <tuple exp> RENAME { <renaming commalist> }
```

The <*tuple exp*> must not be a <*monadic or dyadic other built in tuple op inv*>. The individual <*renaming*>s are effectively executed in parallel.[12]

```
<renaming>
    ::=   <attribute ref> AS <introduced name>
        | PREFIX <character string literal>
              AS <character string literal>
        | SUFFIX <character string literal>
              AS <character string literal>
```

For the syntax of <*character string literal*>, see <*built in scalar type name*>. The <*renaming*> PREFIX *a* AS *b* causes all attributes of the applicable tuple or relation whose name begins with the characters of *a* to be renamed such that their name begins with the characters of *b* instead. The <*renaming*> SUFFIX *a* AS *b* is defined analogously.

```
<tuple extend>
    ::=   EXTEND <tuple exp> : { <attribute assign commalist> }
```

The individual <*attribute assign*>s are effectively executed in parallel. Let *t* be the tuple denoted by the <*tuple exp*>, and let *A1, A2, ..., An* be the attributes of *t*. Every <*attribute assign*>, *AA* say, in the <*attribute assign commalist*> is syntactically identical to an <*assign*> (i.e., a <*scalar assign*>, a <*tuple assign*>, or a <*relation assign*>, as applicable), except that:

- The target of *AA* must be either an <*introduced name*> *N* or an <*attribute target*> *AT*.
- If *N* is specified, it must be distinct from every *Ai* (*i* = 1, 2, ..., *n*). If *AT* is specified, then it must identify, directly or indirectly, some *Ai* (*i* = 1, 2, ..., *n*).
- *AA* is allowed to contain an <*attribute ref*>, *AR* say, wherever a <*selector inv*> would be allowed. If the <*attribute name*> of *AR* is that of some *Ai* (*i* = 1, 2, ..., *n*), then *AR* denotes the corresponding attribute

---

[12] RENAME as defined in the *Manifesto* book used parentheses, not braces, and the individual <*renaming*>s were executed in sequence instead of in parallel. Of course, the effect of sequential execution can always be obtained if desired by nesting, as in, e.g., (*t* RENAME {*A* AS *B*}) RENAME {*B* AS *C*}. Similar remarks apply to certain other operators as well—essentially, all of those operators for which the prose explanation in this chapter includes the phrase "effectively executed in parallel."

value from *t;* otherwise the <*tuple extend*> must be contained within some expression in which the meaning of *AR* is defined.

Steps a. and b. of the definition given for multiple assignment under RM Prescription 21 (see Chapter 1 of the present book) are applied to the <*attribute assign commalist*>. The result of that application is an <*attribute assign commalist*> in which each <*attribute assign*> is of the form

```
X := exp
```

Here each such *X* is either an <*introduced name*> or some *Ai,* and no two distinct <*attribute assign*>s specify the same target. Now:

- Consider the expression

  ```
  EXTEND t : { X := XX }
  ```

  where *X* is an introduced name. (For simplicity, we consider only the case where there is just one <*attribute assign*>. The considerations involved in dealing with more than one are essentially straightforward.) The value of this expression is a tuple identical to *t* except that it has an additional attribute called *X,* with declared type and value as specified by *XX*.

- Alternatively, consider the expression

  ```
  EXTEND t : { Y := YY }
  ```

  where *Y* is some *Ai*. (Again we consider for simplicity only the case where there is just one <*attribute assign*>.) This expression is equivalent to the following:

  ```
  ( EXTEND t : { X := YY } ) { ALL BUT Y } RENAME { X AS Y }
  ```

  Here *X* is an arbitrary <*introduced name*>, distinct from all existing attribute names in *t*.[13]

```
<attribute target>
    ::=   <attribute ref>
        | <attribute THE_ pv ref>

<attribute THE_ pv ref>
    ::=   <THE_ pv name> ( <attribute target> )

<tuple wrap>
    ::=   <tuple exp> WRAP <wrapping>
```

The <*tuple exp*> must not be a <*monadic or dyadic other built in tuple op inv*>.

```
<wrapping>
    ::=   { [ ALL BUT ] <attribute ref commalist> }
                                          AS <introduced name>

<tuple unwrap>
    ::=   <tuple exp> UNWRAP <unwrapping>
```

---

[13] Observe that this latter form of <*tuple extend*> replaces <*tuple substitute*> as defined in the *Manifesto* book (and a similar remark applies to <*extend*>, q.v.). *Note*: The keyword EXTEND is perhaps not the best in the circumstances, but it's hard to find a word that catches the overall sense better and yet is equally succinct.

The *<tuple exp>* must not be a *<monadic or dyadic other built in tuple op inv>*.

```
<unwrapping>
    ::=   <attribute ref>
```

The declared type of the specified attribute must be some tuple type.

```
<dyadic other built in tuple op inv>
    ::=   <dyadic tuple union> | <dyadic tuple compose>

<dyadic tuple union>
    ::=   <tuple exp> UNION <tuple exp>
```

The *<tuple exp>*s must not be *<monadic or dyadic other built in tuple op inv>*s, except that either or both can be another *<dyadic tuple union>*. *Note:* Dyadic tuple INTERSECT and MINUS operators could also be defined if desired. A dyadic tuple XUNION operator could be defined too, but it would be logically equivalent to *<dyadic tuple compose>* (see the production rule immediately following).

```
<dyadic tuple compose>
    ::=   <tuple exp> COMPOSE <tuple exp>
```

The *<tuple exp>*s must not be *<monadic or dyadic other built in tuple op inv>*s (not even another *<dyadic tuple compose>*).

```
<tuple assign>
    ::=   <tuple target> := <tuple exp>
        | <tuple update>

<tuple target>
    ::=   <tuple var ref>
        | <tuple THE_ pv ref>

<tuple THE_ pv ref>
    ::=   <THE_ pv name> ( <scalar target> )
```

The declared type of the *<possrep component>* corresponding to *<THE_ pv name>* must be some tuple type.

```
<tuple update>
    ::=   UPDATE <tuple target> :
                { <attribute assign commalist> }
```

Let *TT* be the *<tuple target>*, and let *A1, A2, ..., An* be the attributes of *TT*. Every *<attribute assign>*, *AA* say, in the *<attribute assign commalist>* is syntactically identical to an *<assign>*, except that:

- The target of *AA* must be an *<attribute target>*, *AT* say.
- *AT* must identify, directly or indirectly, some *Ai* (*i* = 1, 2, ..., *n*).
- *AA* is allowed to contain an *<attribute ref>*, *AR* say, wherever a *<selector inv>* would be allowed. If the *<attribute name>* of *AR* is that of some *Ai* (*i* = 1, 2, ..., *n*), then *AR* denotes the corresponding attribute value from *TT;* otherwise the *<tuple update>* must be contained within some expression in which the meaning of *AR* is defined.

Steps a. and b. of the definition given for multiple assignment under RM Prescription 21 (see Chapter 1 of the present book) are applied to the <*attribute assign commalist*>. The result of that application is an <*attribute assign commalist*> in which each <*attribute assign*> is of the form

```
Ai := exp
```

for some *Ai,* and no two distinct <*attribute assign*>s specify the same target *Ai*. Now consider the <*tuple update*>

```
UPDATE TT : { X := XX }
```

where *X* is some *Ai*. (For simplicity, we consider only the case where there is just one <*attribute assign*>. The considerations involved in dealing with more than one are essentially straightforward.) This <*tuple update*> is equivalent to the following <*tuple assign*>:

```
TT := EXTEND TT : { X := XX }
```

```
<tuple comp>
    ::=   <tuple exp> <tuple comp op> <tuple exp>
        | <tuple exp> ∈ <relation exp>
        | <tuple exp> ∉ <relation exp>
```

Tuple comparisons are a special case of the syntactic category <*bool exp*>. The symbol "∈" denotes the set membership operator; it can be read as *belongs to* or *is a member of* or just *in* or *is in*. The expression $t \notin r$ is semantically equivalent to the expression NOT ($t \in r$).

```
<tuple comp op>
    ::=   = | ≠
```

RELATIONAL OPERATIONS

Note that definitions of the semantics of many (not all) of the operators described in this section can be found in Appendix A of reference [7].

```
<relation nonwith exp>
    ::=   <relation var ref>
        | <relation op inv>
        | ( <relation exp> )
```

```
<relation op inv>
    ::=   <user op inv>
        | <built in relation op inv>
```

```
<built in relation op inv>
    ::=   <relation selector inv>
        | <THE_ op inv>
        | <attribute extractor inv>
        | <project>
        | <n-adic other built in relation op inv>
        | <monadic or dyadic other built in relation op inv>
```

```
<relation selector inv>
    ::=   RELATION [ <heading> ] { <tuple exp commalist> }
        | TABLE_DEE
        | TABLE_DUM
```

If the keyword RELATION is specified, (a) <*heading*> must be specified if <*tuple exp commalist*> is empty; (b) every <*tuple exp*> in the <*tuple exp commalist*> must have the same heading; (c) that heading must be exactly as defined by <*heading*> if <*heading*> is specified. TABLE_DEE and TABLE_DUM are shorthand for RELATION {} {TUPLE {}} and RELATION {} {}, respectively. *Note:* The proposals of reference [9], q.v., classify TABLE_DEE and TABLE_DUM not as <*relation selector inv*>s but as <*relation const ref*>s ("relation constant references") instead.

```
<project>
    ::=   <relation exp> { [ ALL BUT ] <attribute ref commalist> }
```

The <*relation exp*> must not be a <*monadic or dyadic other built in relation op inv*>. *Note:* As mentioned earlier, although we generally have little to say regarding operator precedence, we find it convenient to assign high precedence to projection in particular.

```
<n-adic other built in relation op inv>
    ::=   <n-adic union> | <n-adic disjoint union>
        | <n-adic intersect> | <n-adic join> | <n-adic times>
        | <n-adic xunion> | <n-adic compose>

<n-adic union>
    ::=   UNION [ <heading> ] { <relation exp commalist> }
```

Here (a) <*heading*> must be specified if <*relation exp commalist*> is empty; (b) every <*relation exp*> in the <*relation exp commalist*> must have the same heading; (c) that heading must be exactly as defined by <*heading*> if <*heading*> is specified. The same remarks apply to <*n-adic disjoint union*>, <*n-adic intersect*>, and <*n-adic xunion*>, q.v.

```
<n-adic disjoint union>
    ::=   D_UNION [ <heading> ] { <relation exp commalist> }

<n-adic intersect>
    ::=   INTERSECT [ <heading> ] { <relation exp commalist> }

<n-adic join>
    ::=   JOIN { <relation exp commalist> }

<n-adic times>
    ::=   TIMES { <relation exp commalist> }
```

The expression TIMES{*r1*,*r2*,...,*rn*} ($n \geqslant 0$) is defined if and only if relations *r1, r2, ..., rn* have no attribute names in common, in which case it's semantically equivalent to JOIN{*r1*,*r2*,...,*rn*}.

```
<n-adic xunion>
    ::=   XUNION [ <heading> ] { <relation exp commalist> }
```

Let *r1, r2, ..., rn* ($n \geqslant 0$) be relations all of the same type; then XUNION{*r1*,*r2*,...,*rn*} is a relation of that same type with body consisting just of those tuples *t* that appear in exactly *m* of *r1, r2, ..., rn,* where *m* is odd (and possibly different for different tuples *t*). See reference [6] for further explanation.

```
<n-adic compose>
    ::=   COMPOSE { <relation exp commalist> }
```

Let *r1, r2, ..., rn* ($n \geqslant 0$) be relations; then COMPOSE{*r1*,*r2*,...,*rn*} is shorthand for the projection on {*X*} of JOIN{*r1*,*r2*,...,*rn*}, where {*X*} is all of the attributes of *r1, r2, ..., rn* apart from ones common to at least two of those relations. See reference [6] for further explanation.

```
<monadic or dyadic other built in relation op inv>
    ::=   <monadic other built in relation op inv>
        | <dyadic other built in relation op inv>

<monadic other built in relation op inv>
    ::=   <rename> | <where> | <extend> | <wrap> | <unwrap>
        | <group> | <ungroup> | <tclose>

<rename>
    ::=   <relation exp> RENAME { <renaming commalist> }
```

The *<relation exp>* must not be a *<monadic or dyadic other built in relation op inv>*. The individual *<renaming>*s are effectively executed in parallel.

```
<where>
    ::=   <relation exp> WHERE <bool exp>
```

The *<relation exp>* must not be a *<monadic or dyadic other built in relation op inv>*. Let *r* be the relation denoted by *<relation exp>*. The *<bool exp>* is allowed to contain an *<attribute ref>*, *AR* say, wherever a *<selector inv>* would be allowed. The *<bool exp>* can be thought of as being evaluated for each tuple of *r* in turn. If the *<attribute name>* of *AR* is that of an attribute of *r,* then (for each such evaluation) *AR* denotes the corresponding attribute value from the corresponding tuple of *r;* otherwise the *<where>* must be contained in some expression in which *AR* is defined. *Note:* The *<where>* operator of **Tutorial D** includes the *restrict* operator of relational algebra as a special case.

```
<extend>
    ::=   EXTEND <relation exp> : { <attribute assign commalist> }
```

The individual *<attribute assign>*s are effectively executed in parallel. Let *r* be the relation denoted by the *<relation exp>*, and let *A1, A2, ..., An* be the attributes of *r*. Every *<attribute assign>*, *AA* say, in the *<attribute assign commalist>* is syntactically identical to an *<assign>* (i.e., a *<scalar assign>*, a *<tuple assign>*, or a *<relation assign>*, as applicable), except that:

- The target of *AA* must be either an *<introduced name> N* or an *<attribute target> AT*.
- If *N* is specified, it must be distinct from every *Ai* (*i* = 1, 2, ..., *n*). If *AT* is specified, then it must identify, directly or indirectly, some *Ai* (*i* = 1, 2, ..., *n*).
- *AA* is allowed to contain an *<attribute ref>*, *AR* say, wherever a *<selector inv>* would be allowed. *AA* can be thought of as being executed for each tuple of *r* in turn. If the *<attribute name>* of *AR* is that of some *Ai* (*i* = 1, 2, ..., *n*), then (for each such application) *AR* denotes the corresponding attribute value from the corresponding tuple of *r;* otherwise the *<extend>* must be contained within some expression in which the meaning of *AR* is defined.

Steps a. and b. of the definition given for multiple assignment under RM Prescription 21 (see Chapter 1 of the present book) are applied to the *<attribute assign commalist>*. The result of that application is an *<attribute assign commalist>* in which each *<attribute assign>* is of the form

```
X := exp
```

Here each such *X* is either an *<introduced name>* or some *Ai,* and no two distinct *<attribute assign>*s specify the same target. Now:

- Consider the expression

```
EXTEND r : { X := XX }
```

where *X* is an introduced name. (For simplicity, we consider only the case where there is just one <*attribute assign*>. The considerations involved in dealing with more than one are essentially straightforward.) The value of this expression is a relation with (a) heading the heading of *r* extended with attribute *X* (with declared type as specified by *XX*) and (b) body consisting of all tuples *t* such that *t* is a tuple of *r* extended with a value for attribute *X* that is computed by evaluating the expression *XX* on that tuple of *r*.

- Alternatively, consider the expression

```
EXTEND r : { Y := YY }
```

where *Y* is some *Ai*. (Again we consider for simplicity only the case where there is just one <*attribute assign*>.) This expression is equivalent to the following:

```
( EXTEND r : { X := YY } ) { ALL BUT Y } RENAME { X AS Y }
```

Here *X* is an arbitrary <*introduced name*>, distinct from all existing attribute names in *r*.

```
<wrap>
    ::=   <relation exp> WRAP <wrapping>
```

The <*relation exp*> must not be a <*monadic or dyadic other built in relation op inv*>.

```
<unwrap>
    ::=   <relation exp> UNWRAP <unwrapping>
```

The <*relation exp*> must not be a <*monadic or dyadic other built in relation op inv*>.

```
<group>
    ::=   <relation exp> GROUP <grouping>
```

The <*relation exp*> must not be a <*monadic or dyadic other built in relation op inv*>.

```
<grouping>
    ::=   { [ ALL BUT ] <attribute ref commalist> }
                                       AS <introduced name>
```

```
<ungroup>
    ::=    <relation exp> UNGROUP <ungrouping>
```

The <*relation exp*> must not be a <*monadic or dyadic other built in relation op inv*>.

```
<ungrouping>
    ::=   <attribute ref>
```

The declared type of the specified attribute must be some relation type.

```
<tclose>
    ::=   TCLOSE ( <relation exp> )
```

The <*relation exp*> must not be a <*monadic or dyadic other built in relation op inv*>. Furthermore, it must denote a relation of degree two, and the declared types of the attributes of that relation must both be the same.

```
<dyadic other built in relation op inv>
    ::=   <dyadic union> | <dyadic disjoint union>
        | <dyadic intersect> | <minus> | <included minus>
        | <dyadic join> | <dyadic times> | <dyadic xunion>
        | <dyadic compose> | <matching> | <not matching>
        | <divide> | <summarize>
```

```
<dyadic union>
    ::=   <relation exp> UNION <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s, except that either or both can be another <*dyadic union*>.

```
<dyadic disjoint union>
    ::=   <relation exp> D_UNION <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s, except that either or both can be another <*dyadic disjoint union*>. The operand relations must have no tuples in common.

```
<dyadic intersect>
    ::=   <relation exp> INTERSECT <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s, except that either or both can be another <*dyadic intersect*>.

```
<minus>
    ::=   <relation exp> MINUS <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s.

```
<included minus>
    ::=   <relation exp> I_MINUS <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s. The second operand relation must be included in the first.

```
<dyadic join>
    ::=   <relation exp> JOIN <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s, except that either or both can be another <*dyadic join*>.

```
<dyadic times>
    ::=   <relation exp> TIMES <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s, except that either or both can be another <*dyadic times*>.

```
<dyadic xunion>
    ::=   <relation exp> XUNION <relation exp>
```

The dyadic XUNION operator (“exclusive union”) is essentially symmetric difference as usually understood. The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s, except that either or both can be another <*dyadic xunion*>.

```
<dyadic compose>
    ::=   <relation exp> COMPOSE <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s (not even another <*dyadic compose*>).

```
<matching>
    ::=   <relation exp> MATCHING <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s. The keyword MATCHING can alternatively be spelled SEMIJOIN.

```
<not matching>
    ::=   <relation exp> NOT MATCHING <relation exp>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s. The keywords NOT MATCHING can alternatively be spelled SEMIMINUS.

```
<divide>
    ::=   <relation exp> DIVIDEBY <relation exp> <per>
```

The <*relation exp*>s must not be <*monadic or dyadic other built in relation op inv*>s. *Note:* Elsewhere this book proposes that DIVIDEBY should be dropped. See references [4] and [9].

```
<per>
    ::=   PER ( <relation exp> [ , <relation exp> ] )
```

Reference [1] defines two distinct "divide" operators that it calls the Small Divide and the Great Divide, respectively. In **Tutorial D,** a <*divide*> in which the <*per*> contains just one <*relation exp*> is a Small Divide, a <*divide*> in which it contains two is a Great Divide.

```
<summarize>
    ::=   SUMMARIZE <relation exp> [ <per or by> ] :
                                   { <attribute assign commalist> }
```

The individual <*attribute assign*>s are effectively executed in parallel. Further explanation appears in the prose following the next three production rules. *Note:* Elsewhere this book proposes that SUMMARIZE should be dropped. See references [4] and [9].

```
<per or by>
    ::=   <per>
        | BY { [ ALL BUT ] <attribute ref commalist> }
```

If <*per*> is specified, it must contain exactly one <*relation exp*>. Let *p* be the relation denoted by that <*relation exp*>, let *r* be the relation denoted by the <*relation exp*> immediately following the keyword SUMMARIZE, and let *B1, B2, ..., Bm* and *A1, A2, ..., An* be the attributes of *p* and *r,* respectively. Every *Bi* (*i* = 1, 2, ..., *m*) must be some *Aj* (*j* = 1, 2, ..., *n*). Specifying BY {*Bx,By,...,Bz*} is equivalent to specifying PER (*r*{*Bx,By,...,Bz*}). Omitting <*per or by*> is equivalent to specifying PER (TABLE_DEE).

Every <*attribute assign*>, *AA* say, in the <*attribute assign commalist*> is syntactically identical to an <*assign*> (i.e., a <*scalar assign*>, a <*tuple assign*>, or a <*relation assign*>, as applicable), except that the target must be an <*introduced name*>, distinct from every *Bi* (*i* = 1, 2, ..., *m*), and the source is allowed to contain a <*summary*> wherever a <*selector inv*> would be allowed (see the production rule immediately following). Steps a. and b. of the definition given for multiple assignment under RM Prescription 21 (see Chapter 1 of the present book) are applied to the <*attribute assign commalist*>. The result of that application is an <*attribute assign* *commalist*> in which each <*attribute assign*> is of the form

```
X := exp
```

where *X* is an <*introduced name*> that is distinct from every *Bi,* and no two distinct <*attribute assign*>s specify the same target. Now consider the expression

```
SUMMARIZE r PER ( p ) : { X := SUM ( XX ) }
```

(For simplicity, we consider only the case where there is just one <*attribute assign*>, the source consists of just a single <*summary*> in isolation, and the corresponding <*summary spec*> is SUM; the considerations involved in dealing with other cases are tedious but essentially straightforward.) This <*summarize*> is defined to be logically equivalent to the following:

```
EXTEND ( p ) : { X := SUM ( ( ( r ) MATCHING
               RELATION { TUPLE { B1 B1 , B2 B2 , ... , Bm Bm } } )
                              { ALL BUT B1 , B2 , ... , Bm } , XX }
```

In other words, the value of the <*summarize*> expression is a relation with (a) heading the heading of *p* extended with attribute *X* (with declared type as specified by *XX*) and (b) body consisting of all tuples *t* such that *t* is a tuple of *p* extended with a value *x* for attribute *X*. That value *x* is computed by evaluating the summary *XX* over all tuples of *r* that have the same value for attributes *B1, B2, ..., Bm* as tuple *t* does.

```
<summary>
    ::=   <summary spec> ( [ <integer exp> , ] [ <exp> ] )
```

Let *r* and *p* be as defined under the production rule for <*per or by*>. Then:

- The <*integer exp*> and following comma must be specified if and only if the <*summary spec*> is EXACTLY; the <*exp*> must be specified if and only if the <*summary spec*> is not COUNT. The <*exp*> (if specified) is effectively evaluated for each tuple of *p* in turn. Let *ee* be such an evaluation, and let the corresponding tuple of *p* be *eet*.

- The <*integer exp*> is allowed to contain an <*attribute ref*>, *IAR* say, wherever a <*selector inv*> would be allowed. If the <*attribute name*> of *IAR* is that of some attribute *Bi* (*i* = 1, 2, ..., *m*) of *p,* then (for each evaluation *ee*) *IAR* denotes the corresponding attribute value from tuple *eet;* otherwise the <*summarize*> must be contained within some expression in which the meaning of *AR* is defined.

- The <*exp*> is allowed to contain an <*attribute ref*>, *AR* say, wherever a <*selector inv*> would be allowed. The <*attribute name*> of *AR* can be—and usually is—that of some attribute *Aj* (*j* = 1, 2, ..., *n*) of *r* but must not be that of any attribute *Bi* (*i* = 1, 2, ..., *m*) of *p*. If the <*attribute name*> of *AR* is that of an attribute of *r,* then (for each evaluation *ee*) *AR* denotes the corresponding attribute value from some tuple of *r* that has the same values for *B1, B2, ..., Bm* as tuple *eet* does; otherwise the <*summarize*> must be contained within some expression in which the meaning of *AR* is defined.

- For SUM and AVG, the declared type of <*exp*> must be some type for which the operator "+" is defined; for MAX and MIN, it must be some ordered type; for AND, OR, EQUIV, XOR, and EXACTLY, it must be type BOOLEAN; for UNION, D_UNION, INTERSECT, and XUNION, it must be some relation type.

```
<summary spec>
    ::=   COUNT | SUM | AVG | MAX | MIN
        | AND | OR | EQUIV | XOR | EXACTLY
        | UNION | D_UNION | INTERSECT | XUNION
```

COUNT returns a result of declared type INTEGER; SUM, AVG, MAX, MIN, UNION, D_UNION, INTERSECT, and XUNION return a result of declared type the same as that of the applicable <*exp*>;[14] AND, OR, EQUIV, XOR, and EXACTLY return a result of declared type BOOLEAN.

```
<relation assign>
    ::=   <relation target> := <relation exp>
        | <relation insert>
        | <relation d_insert>
        | <relation delete>
        | <relation i_delete>
        | <relation update>

<relation target>
    ::=   <relation var ref>
        | <relation THE_ pv ref>

<relation THE_ pv ref>
    ::=   <THE_ pv name> ( <scalar target> )
```

The declared type of the <*possrep component*> corresponding to <*THE_ pv name*> must be some relation type. *Note:* Let *rx* be a <*relation exp*> that could appear in the <*virtual relation var def*> that defines some updatable virtual relvar *V* (see references [3] and [8], also Chapter 10 of the present book). Then it would be possible to allow *rx* to serve as a relation pseudovariable also. However, this possibility is not reflected in the grammar defined in this chapter.

```
<relation insert>
    ::=   INSERT <relation target> <relation exp>

<relation d_insert>
    ::=   D_INSERT <relation target> <relation exp>
```

The difference between INSERT and D_INSERT is that, loosely speaking, an attempt to insert a tuple that already exists succeeds with INSERT but fails with D_INSERT. (In other words, INSERT is defined in terms of UNION, while D_INSERT is defined in terms of D_UNION.)

```
<relation delete>
    ::=   DELETE <relation target> <relation exp>
        | DELETE <relation target> [ WHERE <bool exp> ]
```

Let *RT* be a <*relation target*>. Then the <*relation delete*> DELETE *RT* WHERE *bx* is shorthand for the <*relation delete*> DELETE *RT rx,* where the <*relation exp*> *rx* is a <*where*> of the form *RT* WHERE *bx.*

```
<relation i_delete>
    ::=   I_DELETE <relation target> <relation exp>
```

The relation denoted by the <*relation exp*> must be included in the current value of the <*relation target*>.

---

[14] It might be preferable in practice to define AVG in such a way that, e.g., taking the average of a collection of integers returns a rational number. We do not do so here merely for reasons of simplicity.

```
<relation update>
    ::=   UPDATE <relation target> [ WHERE <bool exp> ] :
                 { <attribute assign commalist> }
```

Let *RT* be the <*relation target*>, and let *A1, A2, ..., An* be the attributes of *RT*. The <*bool exp*> is allowed to contain an <*attribute ref*>, *AR* say, wherever a <*selector inv*> would be allowed. The <*bool exp*> can be thought of as being evaluated for each tuple of *RT* in turn. If the <*attribute name*> of *AR* is that of some *Ai* (*i* = 1, 2, ..., *n*), then (for each such evaluation) *AR* denotes the corresponding attribute value from the corresponding tuple; otherwise the <*relation update*> must be contained within some expression in which the meaning of *AR* is defined. Every <*attribute assign*>, *AA* say, in the <*attribute assign commalist*> is syntactically identical to an <*assign*>, except that:

- The target of *AA* must be an <*attribute target*>, *AT* say.
- *AT* must identify, directly or indirectly, some *Ai* (*i* = 1, 2, ..., *n*).
- *AA* is allowed to contain an <*attribute ref*>, *AR* say, wherever a <*selector inv*> would be allowed. *AA* can be thought of as being applied to each tuple of *r* in turn. If the <*attribute name*> of *AR* is that of some *Ai* (*i* = 1, 2, ..., *n*), then (for each such application) *AR* denotes the corresponding attribute value from the corresponding tuple; otherwise the <*relation update*> must be contained within some expression in which the meaning of *AR* is defined.

Steps a. and b. of the definition given for multiple assignment under RM Prescription 21 (see Chapter 1 of the present book) are applied to the <*attribute assign commalist*>. The result of that application is an <*attribute assign commalist*> in which each <*attribute assign*> is of the form

```
Ai := exp
```

for some *Ai,* and no two distinct <*attribute assign*>s specify the same target *Ai*. Now consider the <*relation update*>

```
UPDATE RT WHERE b : { X := XX }
```

where *X* is some *Ai*. (For simplicity, we consider only the case where there is just one <*attribute assign*>. The considerations involved in dealing with more than one are essentially straightforward.) This <*relation update*> is equivalent to the following <*relation assign*>:

```
RT := ( RT WHERE NOT ( b ) )
         UNION
       ( EXTEND RT WHERE b : { X := XX } )
```

```
<relation comp>
    ::=   <relation exp> <relation comp op> <relation exp>
```

Relation comparisons are a special case of the syntactic category <*bool exp*>.

```
<relation comp op>
    ::=   = | ≠ | ⊆ | ⊇ | ⊂ | ⊃
```

The symbols "⊆" and "⊂" denote "subset of" and "proper subset of," respectively; the symbols "⊇" and "⊃" denote "superset of" and "proper superset of," respectively.

RELATIONS AND ARRAYS

The Third Manifesto forbids tuple level retrieval from a relation (in other words, it prohibits anything analogous to SQL's FETCH via a cursor). But **Tutorial D** does allow a relation to be mapped to a one-dimensional array (of tuples), so an effect somewhat analogous to such tuple level retrieval can be obtained, if desired, by first performing such a mapping and then iterating over the resulting array.[15] But we deliberately adopt a very conservative approach to this part of the language. A fully orthogonal language would support arrays as "first class citizens"—implying support for a general ARRAY type generator, and arrays of any number of dimensions, and array expressions, and array assignment, and array comparisons, and so on. However, to include such extensive support in **Tutorial D** would complicate the language unduly and might well obscure more important points. For simplicity, therefore, we include only as much array support here as seems absolutely necessary; moreover, most of what we do include is deliberately "special cased." Note in particular that we don't define a syntactic category called *<array type spec>*.

```
<array var def>
    ::=   VAR <array var name> ARRAY <tuple type spec>
```

Let *A* be a **Tutorial D** array variable; then the value of *A* at any given time is a one-dimensional array containing zero or more tuples all of the same type. Let the values of *A* at times *t1* and *t2* be *a1* and *a2,* respectively. Then *a1* and *a2* need not necessarily contain the same number of tuples, and *A*'s upper bound thus varies with time (the lower bound, by contrast, is always zero).[16] Note that the only way *A* can acquire a new value is by means of a *<relation get>* (see below); in practice, of course, additional mechanisms will be desirable, but no such mechanisms are specified here.

```
<relation get>
    ::=   LOAD <array target> FROM <relation exp>
                               ORDER ( <order item commalist> )

<array target>
    ::=   <array var ref>

<array var ref>
    ::=   <array var name>
```

Points arising:

- Tuples from the relation denoted by *<relation exp>* are loaded into the array variable designated by *<array target>* in the order defined by the ORDER specification. If *<order item commalist>* is empty, tuples are loaded in an implementation defined order.
- The headings associated with *<array target>* and *<relation exp>* would normally be the same. But it would be possible, and perhaps desirable, to allow the former to be a proper subset of the latter. Such a feature could allow the sequence in which tuples are loaded into the array variable to be defined in terms of attributes whose values aren't themselves to be retrieved—thereby allowing, e.g., retrieval of employee numbers and names in salary order without at the same time actually retrieving those salaries.

---

[15] By contrast, in accordance with RM Proscription 7 (see Chapter 1 of the present book), **Tutorial D** supports nothing comparable to SQL's tuple at a time update operators—i.e., UPDATE or DELETE "WHERE CURRENT OF *cursor*"—at all.

[16] Chapter 7 of the *Manifesto* book includes a coding example in which the lower bound is assumed to be one.

- LOAD is really assignment, of a kind (in particular, it has the effect of replacing whatever value the target previously had). However, we deliberately choose not to use assignment syntax for it because it effectively involves an implicit type conversion (i.e., a coercion) between a relation and an array. As a general rule, we prefer not to support coercions at all; in the case at hand, therefore, we prefer to define a new operation (LOAD), with operands that are explicitly defined to be of different types, instead of relying on conventional assignment plus coercion.

```
<order item>
    ::=   <direction> <attribute ref>
```

The attribute identified by <*attribute ref*> must be of some ordered type. A useful extension in practice might be to allow <*scalar exp*> in place of <*attribute ref*> here.

```
<direction>
    ::=   ASC | DESC

<relation set>
    ::=   LOAD <relation target> FROM <array var ref>
```

The array identified by <*array var ref*> must not include any duplicate tuples.

We also need a new kind of <*tuple nonwith exp*> and an <*array cardinality*> operator (a special case of <*integer exp*>):

```
<tuple nonwith exp>
    ::=   ... all previous possibilities, together with:
        | <array var ref> ( <subscript> )
```

We remark that it might be preferable in practice for subscripts to be enclosed in square brackets instead of parentheses. As it is, a <*tuple nonwith exp*> of the form *A*(*I*) is syntactically indistinguishable from an invocation of an operator called *A* with a single argument *I* of type INTEGER.

```
<subscript>
    ::=   <integer exp>

<array cardinality>
    ::=   COUNT ( <array var ref> )
```

STATEMENTS

```
<statement>
    ::=   <statement body> ;

<statement body>
    ::=   <with statement body>
        | <nonwith statement body>

<with statement body>
    ::=   WITH ( <name intro commalist> ) : <statement body>
```

Let *WSB* be a <*with statement body*>, and let *NIC* and *SB* be the <*name intro commalist*> and the <*statement body*>, respectively, immediately contained in *WSB*. The individual <*name intro*>s in *NIC* are evaluated in sequence as written. By definition, each such <*name intro*> immediately contains an <*introduced name*> and an <*exp*>. Let *NI* be one of those <*name intro*>s, and let the <*introduced name*> and the <*exp*> immediately contained in *NI* be *N* and *X,* respectively. Then *N* denotes the value obtained by evaluating *X,* and it can appear subsequently in *WSB* wherever the expression (*X*)—i.e., *X* in parentheses—would be allowed.

```
<nonwith statement body>
    ::=   <previously defined statement body commalist>
        | <begin transaction> | <commit> | <rollback>
        | <call> | <return> | <case> | <if> | <do> | <while>
        | <leave> | <no op> | <compound statement body>
```

```
<previously defined statement body>
    ::=   <assignment>
        | <user op def> | <user op drop>
        | <user scalar type def> | <user scalar type drop>
        | <scalar var def> | <tuple var def>
        | <relation var def> | <relation var drop>
        | <constraint def> | <constraint drop>
        | <array var def> | <relation get> | <relation set>
```

```
<begin transaction>
    ::=   BEGIN TRANSACTION
```

BEGIN TRANSACTION can be issued when a transaction is in progress. The effect is to suspend the current transaction and to begin a new ("child") transaction. COMMIT or ROLLBACK terminates the transaction most recently begun, thereby resuming the suspended "parent" transaction, if any. *Note:* An industrial strength **D** might usefully allow BEGIN TRANSACTION to assign a name to the transaction in question and then require COMMIT and ROLLBACK to reference that name explicitly. However, we choose not to specify any such facilities here.

```
<commit>
    ::=   COMMIT
```

```
<rollback>
    ::=   ROLLBACK
```

```
<call>
    ::=   CALL <user op inv>
```

The user defined operator being invoked must be an update operator specifically. Arguments corresponding to parameters that are subject to update must be specified as *<scalar target>*s, *<tuple target>*s, or *<relation target>*s, as applicable.

```
<return>
    ::=   RETURN [ <exp> ]
```

A *<return>* is permitted only within a *<user read-only op def>* or a *<user update op def>*. The *<exp>* is required in the former case and prohibited in the latter. *Note:* A *<user update op def>* need not contain a *<return>* at all, in which case an implicit *<return>* is executed when the END OPERATOR is reached.

```
<case>
    ::=   CASE ; <when spec list> [ ELSE <statement> ]
          END CASE
```

```
<when spec>
    ::=   WHEN <bool exp> THEN <statement>
```

```
<if>
    ::=   IF <bool exp> THEN <statement> [ ELSE <statement> ]
          END IF

<do>
    ::=   [ <statement name> : ]
          DO <scalar var ref> := <integer exp> TO <integer exp> ;
             <statement>
          END DO

<while>
    ::=   [ <statement name> : ]
          WHILE <bool exp> ;
             <statement>
          END WHILE

<leave>
    ::=   LEAVE <statement name>
```

A variant of *<leave>* that merely terminates the current iteration of the loop and begins the next might be useful in practice.

```
<no op>
    ::=   ... zero or more "white space" characters

<compound statement body>
    ::=   BEGIN ; <statement list> END
```

One final point to close this section: In other writings we often make use of *end of statement, statement boundary,* and similar expressions to refer to the time when integrity checking is done, among other things. In such contexts, *statement* is to be understood, in **Tutorial D** terms, to mean a *<statement>* that contains no other *<statement>*s nested syntactically inside itself; i.e., it isn't a *<case>*, *<if>*, *<do>*, *<while>*, or compound statement. In other words (loosely): Integrity checking is done at semicolons.

RECENT LANGUAGE CHANGES

As mentioned in the introduction to this chapter, there are a number of differences between **Tutorial D** as defined herein and the version defined in Chapter 5 of the *Manifesto* book. For the benefit of readers who might be familiar with that earlier version, we summarize the main differences here.

- INT, RAT, and BOOL have been introduced as synonyms for INTEGER, RATIONAL, and BOOLEAN, respectively; REL and TUP have been introduced as synonyms for RELATION and TUPLE, respectively; DEE and DUM have been introduced as abbreviations for TABLE_DEE and TABLE_DUM, respectively.
- Scalar types now have an associated example value. Example values for the system defined scalar types INTEGER, RATIONAL, CHARACTER, and BOOLEAN have been defined.
- Those example values are used as a basis for defining the initial value for a scalar or tuple variable that has no such explicitly defined value.
- Scalar types can now be defined to be ORDINAL, ORDERED, or neither. Justification for this revision can be found in the section "Recent *Manifesto* Changes" in Chapter 1.

- Specifying an empty <*key def list*> for a real relvar or an application relvar is now interpreted to mean the entire heading is a key for the pertinent relvar.
- The <*name intro commalist*> in WITH is now enclosed in parentheses—not braces, because the individual <*name intro*> are executed in sequence, not in parallel—and those <*name intro*>s now use assignment-style syntax. WITH is now allowed on statements as well as expressions.
- The second argument to an aggregate operator invocation—or third, in the case of EXACTLY—is now specified as a general <*exp*> instead of just a simple <*attribute ref*>, and a similar change has been made to SUMMARIZE. *Note:* This facility is particularly useful in connection with the aggregate operator AND and what are sometimes called "tuple constraints." A tuple constraint is a constraint that can be checked for a given tuple in isolation (i.e., without having to inspect any other tuples in the pertinent relvar and without having to examine any other relvars). Such constraints take the general form "For all tuples in *rx, x* must be true." Previously, we would typically have had to express such a constraint as "The set of tuples in *rx* for which *rx* is not true must be empty"; but the extended form of the AND aggregate operator lets us express it a little more directly. For example, suppose the familiar suppliers-and-parts database is subject to the constraint that supplier status values must always be positive. Old style:

  ```
  CONSTRAINT ... IS_EMPTY ( S WHERE NOT ( STATUS > 0 ) ) ;
  ```

  New style:

  ```
  CONSTRAINT ... AND ( S , STATUS > 0 ) ;
  ```

- The syntax of *n*-adic aggregate operator invocations has been clarified.
- The <*agg op name*>s and <*summary spec*>s ALL and ANY (previously supported as alternative spellings for AND and OR, respectively) were always somewhat error prone and have been dropped.[17]
- The <*summary spec*>s COUNTD, SUMD, AVGD, and EXACTLYD have been dropped. EQUIV has been added to both <*summary spec*> and <*agg op name*>.
- The individual <*renaming*>s in a RENAME invocation are now executed in parallel instead of in sequence, and analogous remarks apply to EXTEND and SUMMARIZE. Braces are now used in all of these operators instead of parentheses.
- The syntax of EXTEND and SUMMARIZE has been revised to use assignment-style syntax.[18] The functionality of the read-only UPDATE or "substitute" operators is now provided by extended forms of the corresponding EXTEND operators (and those "substitute" operators have been dropped).
- The GROUP operator now takes just a single <*grouping*> instead of a <*grouping commalist*>. The reason is that in some cases the order in which individual <*grouping*>s are evaluated in a "multiple grouping" affects the overall result, and we didn't want to have to prescribe a specific evaluation sequence. Similar

---

[17] We are however actively considering FORALL and EXISTS as possible replacements. These keywords work quite well as <*agg op name*>s, less well as <*summary spec*>s. But we're proposing elsewhere (see Chapter 16) that SUMMARIZE be dropped anyway, so this latter objection is perhaps not very significant.

[18] Several writers have criticized this particular revision on the grounds that assignments as such cause "a change in state"—i.e., a change to some variable visible to the user—and EXTEND and SUMMARIZE don't. This observation is clearly correct. But every alternative syntax we've investigated seems to suffer from drawbacks of its own; therefore, given that any such alternative would require rather major surgery on this chapter (surgery for perhaps marginal gain at that), we've decided for the time being to let sleeping dogs lie.

remarks apply to UNGROUP, WRAP, and UNWRAP.

- As previously noted, the commalist of <*assign*>s (of various kinds) in EXTEND and SUMMARIZE, and UPDATE is now enclosed in braces; it is also preceded by a colon. Similar remarks apply to UPDATE (all forms).
- Support for dyadic and *n*-adic cartesian product (TIMES), dyadic and *n*-adic exclusive union (XUNION), *n*-adic COMPOSE, and included minus (I_MINUS) has been added.
- Support for *n*-adic tuple UNION has been added.
- The argument expression in TCLOSE is now enclosed in parentheses.
- Support for D_INSERT and I_DELETE has been added.
- Arrays now always have a lower bound of zero.
- The syntax of <*statement*> has been extended to allow a commalist of any number of <*previously defined statement body*>s preceding the semicolon. Thus, for example, any number of variables, or any number of types, or any number of constraints, can all be defined or destroyed "simultaneously." (Multiple assignment, which was already included in the version of **Tutorial D** defined in the *Manifesto* book, is now just a special case and thus no longer really requires separate production rules of its own.) *Note:* It would be remiss of us not to point out that there's a small unresolved syntax problem in this area, though. One form of <*previously defined statement body*> is a <*user op def*>. But the syntax of a <*user op def*> includes nested semicolons, and those semicolons might be thought to clash with the semicolon that terminates the overall <*statement*>. Possible fixes for this problem include the following:

    1. Do nothing—those semicolons don't actually lead to any syntactic ambiguity, they're just intrusive and awkward.
    2. Require <*user op def*>s to be enclosed in, say, parentheses when they appear in this particular context.
    3. Don't allow <*user op def*>s to appear in this context at all. This fix is probably the least attractive, since it would certainly constitute a violation of orthogonality, and in fact there might be a genuine requirement to be able to define several operators simultaneously (e.g., in cases of mutual recursion).

In addition to the foregoing, many syntactic category names and production rules have been revised (in some cases extensively). However, those revisions in themselves are not intended to induce any changes in the language being defined.

ACKNOWLEDGMENTS

We would like to acknowledge Adrian Hudnott's careful review of an earlier version of this chapter and his several contributions to the final version.

REFERENCES AND BIBLIOGRAPHY

1. Hugh Darwen and C. J. Date: "Into the Great Divide," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

2. C. J. Date: *An Introduction to Database Systems* (8th edition). Boston, Mass.: Addison-Wesley (2004).

3. C. J. Date: "The Logic of View Updating," in *Logic and Databases: The Roots of Relational Theory*. Victoria, B.C.: Trafford Publishing (2007). See *www.trafford.com/07-0690*. See also Chapter 10 of the present book.

4. C. J. Date: "A Brief History of the Relational Divide Operator" (Chapter 12 of the present book).

5. C. J. Date: "Inclusion Dependencies and Foreign Keys" (Chapter 13 of the present book).

6. C. J. Date: "*N*-adic vs. Dyadic Operators: An Investigation" (Chapter 15 of the present book).

7. C. J. Date and Hugh Darwen: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition). Boston, Mass.: Addison-Wesley (2006).

8. C. J. Date and Hugh Darwen: "View Updating" (Appendix E of reference [7]).

9. C. J. Date and Hugh Darwen: "Toward an Industrial Strength Dialect of **Tutorial D**" (Chapter 16 of the present book).

10. C. J. Date and Hugh Darwen: "Extending Tutorial D to Support the Inheritance Model" (Chapter 21 of the present book).

APPENDIX A: A REMARK ON SYNTAX

As you might have noticed, the syntax of operator invocations in **Tutorial D** isn't very consistent. To be specific:

- User defined operators use a prefix style, with positional argument/parameter matching.
- System defined operators, by contrast, sometimes use an infix style ("+", "=", MINUS, etc.), sometimes a prefix style (MAX, EXACTLY, *n*-adic JOIN, etc.).
- Some of those system defined operators rely on positional argument/parameter matching ("+", MINUS, AVG, EXACTLY, etc.), while others don't ("=", *n*-adic JOIN, etc.).
- Some but not all of those system defined operators that rely on positional matching use parentheses to enclose their arguments, while those that don't use braces.
- Some operators seem to use a mixture of prefix and infix styles (SUMMARIZE, DIVIDEBY, etc.), or even a wholly private style of their own (project, THE_ operators, CASE, CAST, etc.).
- Finally, it could be argued that reliance on ordinal position for argument/parameter matching violates the spirit, if not the letter, of RM Proscription 1 (which, as explained in Chapter 1 of the present book, prohibits the use of ordinal position to distinguish the attributes of a relation)—especially in the case of scalar selectors, where the sequence of defining parameters (in the corresponding possrep definition) shouldn't matter but does.

Given all of the above, the possibility of adopting a more uniform style seems worth exploring. Now, we deliberately did no such thing earlier in this chapter because we didn't want **Tutorial D** to look even more outlandish than it might do already. Now, however, we can at least offer some thoughts on the subject. The obvious approach would be to do both of the following:

- Permit (if not mandate) a prefix style for everything

- Perform argument/parameter matching on the basis of names instead of position

  In the case of scalar selectors, for example, we might propose

```
CARTESIAN { Y 2.5, X 7.0 }
```

as a possible replacement for

```
CARTESIAN ( 7.0, 2.5 )
```

Note in particular that the parentheses have been replaced by braces; the assumption in the example is that CARTESIAN ("cartesian coordinates") is a possible representation for a user defined scalar type called, perhaps, POINT. In other words, the suggestion is that operator invocations in general should take the form

```
<op name> { <argument spec commalist> }
```

where *<op name>* identifies the operator in question and *<argument spec>* takes the form

```
<parameter name> <argument exp>
```

or even, possibly,

```
<parameter name> := <argument exp>
```

There are some difficulties, however. First, this new prefix style seems clumsier than the old one in the common special case in which the operator takes just one parameter, as with, e.g., SIN, COS, and (usually) COUNT. Second, some common operators (e.g., "+", "=", ":=") have names that do not abide by the usual rules for forming identifiers. Third, system defined operators, at least as currently defined, have no user known parameter names.

Now, we could perhaps fix this last problem by introducing a convention according to which those names are simply defined to be P1, P2, P3, etc., thus making (e.g.) expressions like this one valid:

```
JOIN { P1 r1 , P2 r2 , P3 r3 , ... , P50 r50 }
```

Again, however, the new syntax in this particular case seems clumsier than before, since JOIN is associative and the order in which the arguments are specified makes no difference.

Another difficulty arises in connection with examples like this one:

```
MINUS { P1 r1 , P2 r2 }
```

Here it becomes important to know which parameter is P1 and which P2 (since *r1* MINUS *r2* and *r2* MINUS *r1* aren't equivalent, in general). Some additional apparatus would be required to communicate such information to the user.

APPENDIX B: EMPTY LISTS AND COMMALISTS

For purposes of reference, in this appendix we (a) repeat the production rules for those **Tutorial D** constructs that include either lists or commalists and (b) explain in each case whether the list or commalist in question is allowed to be empty and, if so, what the significance is. *Note:* In some cases, the significance was in fact explained in the body of the chapter, but we repeat it here for convenience.

```
<user update op def>
    ::=   OPERATOR <user op name> ( <parameter def commalist> )
             UPDATES { [ ALL BUT ] <parameter name commalist> } ;
                <statement>
          END OPERATOR
```

If and only if the <*parameter def commalist*> is empty, then the update operator being defined is niladic and must always be invoked with an empty <*argument exp commalist*>. Also, let the UPDATES specification (either form) identify exactly *n* parameters as being subject to update. If the update operator being defined is niladic, then *n* must be zero. If *n* is zero, then the operator being defined is still an update operator, but it isn't allowed to assign to any of its parameters (it might, however, assign to some "global variable").

```
<user read-only op def>
    ::=   OPERATOR <user op name> ( <parameter def commalist> )
             RETURNS <type spec> ;
                <statement>
          END OPERATOR
```

If and only if the <*parameter def commalist*> is empty, then the read-only operator being defined is niladic and must always be invoked with an empty <*argument exp commalist*>.

```
<user op inv>
    ::=   <user op name> ( <argument exp commalist> )
```

If the <*argument exp commalist*> is empty, then the operator being invoked must be niladic and vice versa.

```
<scalar with exp>
    ::=   WITH ( <name intro commalist> ) : <scalar exp>

<tuple with exp>
    ::=   WITH ( <name intro commalist> ) : <tuple exp>

<relation with exp>
    ::=   WITH ( <name intro commalist> ) : <relation exp>
```

For all expressions *x,* the expression WITH () : *x* is logically equivalent to the expression *x* (regardless of whether *x* is a scalar, tuple, or relation expression).

```
<assignment>
    ::=   <assign commalist>
```

An <*assignment*> with an empty <*assign commalist*> is a <*no op*>.

```
<n-adic bool op name> { <bool exp commalist> }
```

AND {} and EQUIV {} both return TRUE, OR {} and XOR {} both return FALSE.

```
EXACTLY ( <integer exp> , { <bool exp commalist> } )
```

EXACTLY (*n,*{}) returns FALSE unless *n* = 0, in which case it returns TRUE.

```
<user scalar root type def>
    ::=   TYPE <user scalar type name>
                [ <ordering> ] <possrep def list>
                  INIT ( <literal> )
```

The <*possrep def list*> is allowed to be empty if and only if type inheritance is supported, in which case, if the <*possrep def list*> is in fact empty, then the scalar root type being invoked must be a dummy type and vice versa. See Chapter 21 for further explanation.

```
<possrep def>
    ::=   POSSREP [ <possrep name> ]
                  { <possrep component def commalist>
                             [ <possrep constraint def> ] }
```

If the <*possrep component def commalist*> is empty, then the scalar root type being defined has at most one value—in fact, exactly one value, since there are no user defined empty scalar types (see RM Prescription 1).

```
<heading>
    ::=   { <attribute commalist> }
```

The (sole) heading with an empty <*attribute commalist*> is the heading of a tuple or relation of degree zero (in particular, it's the heading for TABLE_DEE and TABLE_DUM).

```
<real relation var def>
    ::=   VAR <relation var name> <real or base>
               <relation type or init value> <key def list>
```

To repeat from the body of the chapter: An empty <*key def list*> is equivalent to a <*key def list*> of the form KEY {ALL BUT}.

```
<key def>
    ::=   KEY { [ ALL BUT ] <attribute ref commalist> }
```

The <*key def*> KEY {} implies that the pertinent relvar can never contain more than one tuple. The <*key def*> KEY {ALL BUT} implies that every relation of the pertinent type is a legitimate value for the pertinent relvar (unless prevented by some further constraint, of course).

```
<virtual relation var def>
    ::=   VAR <relation var name> VIRTUAL
               ( <relation exp> ) <key def list>
```

To repeat from the body of the chapter: An empty <*key def list*> is equivalent to a <*key def list*> that contains exactly one <*key def*> for each key that can be inferred by the system from <*relation exp*>.

```
<application relation var def>
    ::=   VAR <relation var name> <private or public>
               <relation type or init value> <key def list>
```

To repeat from the body of the chapter: An empty <*key def list*> is equivalent to a <*key def list*> of the form KEY {ALL BUT}.

```
CASE <when def list> [ ELSE <scalar exp> ] END CASE
```

If ELSE <*scalar exp*> is omitted, the <*when def list*> must not be empty. The expression

```
CASE ELSE x END CASE
```

is equivalent to *x*.

```
<scalar selector inv>
    ::=   <possrep name> ( <argument exp commalist> )
```

An empty <*argument exp commalist*> is allowed (in fact, required) if and only if the <*possrep def*> identified by <*possrep name*> has an empty <*possrep component def commalist*>—in which case the <*scalar selector inv*> returns the sole value of the applicable type.

```
<n-adic count etc>
    ::=   <agg op name> { <exp commalist> }
```

COUNT {} and SUM_*T* {} return zero (of type INTEGER in the case of COUNT and type *T* in the case of SUM); MAX_*T* {} and MIN_*T* {} return the smallest value and the largest value, respectively, of type *T;* AVG {} raises an exception.

```
<scalar update>
    ::=   UPDATE <scalar target> :
               { <possrep component assign commalist> }
```

The <*scalar update*> UPDATE *ST* : {}, where *ST* is a <*scalar target*>, is equivalent to the <*scalar assign*> *ST* := *ST*.

```
<tuple selector inv>
    ::=   TUPLE { <tuple component commalist> }
```

The <*tuple selector inv*> TUPLE {} denotes the 0-tuple.

```
<tuple project>
    ::=   <tuple exp> { [ ALL BUT ] <attribute ref commalist> }
```

The <*tuple project*> *tx* {}, where *tx* is a <*tuple exp*>, returns the 0-tuple. The <*tuple project*> *tx* {ALL BUT} is equivalent to *tx*.

```
<n-adic tuple union>
    ::=   UNION { <tuple exp commalist> }
```

The <*n-adic tuple union*> UNION {} returns the 0-tuple.

```
<tuple rename>
    ::=   <tuple exp> RENAME { <renaming commalist> }
```

The <*tuple rename*> *tx* RENAME {}, where *tx* is a <*tuple exp*>, is equivalent to *tx*.

```
<tuple extend>
    ::=   EXTEND <tuple exp> : { <attribute assign commalist> }
```

The <*tuple extend*> EXTEND *tx* : {}, where *tx* is a <*tuple exp*>, is equivalent to *tx*.

```
<wrapping>
    ::=   { [ ALL BUT ] <attribute ref commalist> }
                                   AS <introduced name>
```

The <*tuple wrap*> *tx* WRAP {} AS *A*, where *tx* is a <*tuple exp*>, is equivalent to EXTEND *tx* : {*A* := {}}; the <*tuple wrap*> *tx* WRAP {ALL BUT} AS *A*, where again *tx* is a <*tuple exp*>, is equivalent to EXTEND *tx* :

{*A* := {*X*}}, where {*X*} is all of the attributes of *tx*. Analogous remarks apply to the <*wrap*> *rx* WRAP {} AS *A* and the <*wrap*> *rx* WRAP {ALL BUT} AS *A*, where *rx* is a <*relation exp*>.

```
<tuple update>
    ::=   UPDATE <tuple target> :
                 { <attribute assign commalist> }
```

The <*tuple update*> UPDATE *TT* : {}, where *TT* is a <*tuple target*>, is equivalent to the <*tuple assign*> *TT* := *TT*.

```
<relation selector inv>
    ::=   RELATION [ <heading> ] { <tuple exp commalist> }
        | TABLE_DEE
        | TABLE_DUM
```

If an empty <*tuple exp commalist*> is specified, then <*heading*> must be specified, and the <*relation selector inv*> returns the empty relation with the specified heading.

```
<project>
    ::=   <relation exp> { [ ALL BUT ] <attribute ref commalist> }
```

The <*project*> *rx* {}, where *rx* is a <*relation exp*>, returns TABLE_DUM if *rx* is empty and TABLE_DEE otherwise. The <*project*> *rx* {ALL BUT} is equivalent to *rx*.

```
<n-adic union>
    ::=   UNION [ <heading> ] { <relation exp commalist> }
```

If the <*relation exp commalist*> is empty, then <*heading*> must be specified, and the <*n-adic union*> returns the empty relation with the specified heading.

```
<n-adic disjoint union>
    ::=   D_UNION [ <heading> ] { <relation exp commalist> }
```

If the <*relation exp commalist*> is empty, then <*heading*> must be specified, and the <*n-adic disjoint union*> returns the empty relation with the specified heading.

```
<n-adic intersect>
    ::=   INTERSECT [ <heading> ] { <relation exp commalist> }
```

If the <*relation exp commalist*> is empty, then <*heading*> must be specified, and the <*n-adic intersect*> returns the universal relation with the specified heading (i.e., the relation whose body contains every tuple with the specified heading). In practice, the implementation might want to outlaw, or at least flag, any expression that requires such a value to be materialized.

```
<n-adic join>
    ::=   JOIN { <relation exp commalist> }
```

JOIN {} returns TABLE_DEE.

```
<n-adic times>
    ::=   TIMES { <relation exp commalist> }
```

TIMES {} returns TABLE_DEE.

```
<n-adic xunion>
    ::=   XUNION [ <heading> ] { <relation exp commalist> }
```

If the <*relation exp commalist*> is empty, then <*heading*> must be specified, and the <*n-adic xunion*> returns the empty relation with the specified heading.

```
<n-adic compose>
    ::=   COMPOSE { <relation exp commalist> }
```

COMPOSE {} returns TABLE_DEE.

```
<rename>
    ::=   <relation exp> RENAME { <renaming commalist> }
```

The <*rename*> *rx* RENAME {}, where *rx* is a <*relation exp*>, is equivalent to *rx*.

```
<extend>
    ::=   EXTEND <relation exp> : { <attribute assign commalist> }
```

The <*extend*> EXTEND *rx* : {}, where *rx* is a <*relation exp*>, is equivalent to *rx*.

```
<grouping>
    ::=   { [ ALL BUT ] <attribute ref commalist> }
                                    AS <introduced name>
```

The <*group*> *rx* GROUP {} AS *A*, where *rx* is a <*relation exp*>, is equivalent to EXTEND *rx* : {*A* := TABLE_DEE}; the <*group*> *rx* GROUP {ALL BUT} AS *A*, where again *rx* is a <*relation exp*>, is equivalent to EXTEND *rx*{} : {*A* := *rx*}.

```
<summarize>
    ::=   SUMMARIZE <relation exp> [ <per or by> ] :
                                { <attribute assign commalist> }
```

The <*summarize*> SUMMARIZE *rx* PER (*px*) : {}, where *rx* and *px* are <*relation exp*>s, is equivalent to *px*. The <*summarize*> SUMMARIZE *rx* BY {*X*} : {}, where *rx* is a <*relation exp*>, is equivalent to *rx* {*X*}.

```
<per or by>
    ::=   <per>
        | BY { [ ALL BUT ] <attribute ref commalist> }
```

The <*summarize*> SUMMARIZE *rx* BY {} : {...}, where *rx* is a <*relation exp*>, is equivalent to SUMMARIZE *rx* PER (*rx* {}) : {...}. The <*summarize*> SUMMARIZE *rx* BY {ALL BUT} : {...}, where again *rx* is a <*relation exp*>, is equivalent to SUMMARIZE *rx* PER (*rx*) : {...}.

```
<relation update>
    ::=   UPDATE <relation target> [ WHERE <bool exp> ] :
                { <attribute assign commalist> }
```

The <*relation update*> UPDATE *RT* [WHERE *bx*] : {}, where *RT* is a <*relation target*> and *bx* is a <*bool exp*>, is equivalent to the <*relation assign*> *RT* := *RT*.

```
<relation get>
    ::=   LOAD <array target> FROM <relation exp>
                              ORDER ( <order item commalist> )
```

The <*relation get*> LOAD *AT* FROM *rx* ORDER (), where *AT* is an <*array target*> and *rx* is a <*relation exp*>, causes tuples from the relation denoted by *rx* to be loaded into the array variable designated by *AT* in an implementation defined order.

```
<with statement body>
    ::=   WITH ( <name intro commalist> ) : <statement body>
```

The <*with statement body*> WITH () : *S,* where *S* is a <*statement body*>, is logically equivalent to the <*statement body*> *S*.

```
<nonwith statement body>
    ::=   <previously defined statement body commalist>
        | <begin transaction> | <commit> | <rollback>
        | <call> | <return> | <case> | <if> | <do> | <while>
        | <leave> | <no op> | <compound statement body>
```

The <*nonwith statement body*> consisting of an empty <*previously defined statement body commalist*> is a <*no op*>.

```
<case>
    ::=   CASE ; <when spec list> [ ELSE <statement> ]
          END CASE
```

The statement

```
CASE ; ELSE S ; END CASE ;
```

(where *S* is a <*statement body*>) is equivalent to

```
S ;
```

The statement

```
CASE ; END CASE ;
```

is equivalent to a <*no op*>.

```
<compound statement body>
    ::=   BEGIN ; <statement list> END
```

The compound statement

```
BEGIN ; END ;
```

is equivalent to a <*no op*>.