Chapter 5

T u t o r i a l D

INTRODUCTION

Tutorial D is a computationally complete programming language with fully integrated database functionality. It is deliberately not meant to be "industrial strength"; rather, it is a "toy" language, whose principal purpose is to serve as a teaching vehicle. As a consequence, many features that would be required in an industrial-strength language are intentionally omitted. (Extending **Tutorial D** to incorporate such features (thereby turning it into what might be called **Industrial D**) could be a worthwhile project.) For example, there is no support for any of the following:

- Sessions and connections
- Any form of communication with the outside world (I/O facilities, etc.)
- Exception handling and feedback information

In connection with this last point, however, we should at least say that we expressly do not want a form of exception-handling that requires the user to pass a feedback argument on each and every operator invocation (such an approach would effectively force all operators to be update operators).

For obvious reasons, there is also no support for any of the items listed in the subsection "Topics Deliberately Omitted" in Chapter 1 (security and authorization, triggered procedures, and so forth). Nor is there any support for type inheritance; however, extensions to deal with this latter topic are described in Part IV of this book.

Date and Darwen, DATABASES TYPES & THE RELATIONAL MODEL, pp. 93-125,

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

- Language characters, identifiers, scope of names, etc.
- Reserved words (if any), comments,[1] 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)

On the other hand, the language is meant to be well designed, as far as it goes. Indeed, it must be——for otherwise it would not be a valid **D,** since it would violate RM Prescription 26 (which requires every **D** to be constructed according to principles of good language design).

As already noted, **Tutorial D** is computationally complete, meaning that entire applications can be written in the language; it is not just a "data sublanguage" that relies on some host language to provide the necessary computational capabilities. In accordance with the assumptions spelled out in Chapter 1, moreover, it is also (like most languages currently in widespread use) imperative in style——though it is worth mentioning that the "data sublanguage" portion, being based as it is on relational algebra, can in fact be regarded as a functional language if considered in isolation.[*2] In practice we would hope that this portion of the language 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 [76].

Tutorial D is a relational language, of course, but in some respects it can be regarded as an object language as well. For one thing, it supports the concept of single-level storage (see OO Very Strong Suggestion 3). 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 is no reliance on a host language, there is no "impedance mismatch" between the types available inside the database and those available outside (i.e., there is no need to map between the arbitrarily complex types used in the database and the probably rather simple types provided by

[1] In our examples we show comments as text strings bracketed by "/*" and "*/" delimiters.

[2] More precisely, the read-only features of the data sublanguage portion can be so regarded. *Note:* It is well known that a relational data sublanguage can be based on either relational algebra or relational calculus. It is also well known that an algebraic style is intuitively preferable for some tasks, a calculus style for others. As already mentioned, **Tutorial D** uses an algebraic style, for definiteness.

some conventional host language).[3] In other words, we agree with the object community's complaint that there is a serious problem in trying to build an interface between a DBMS that allows user-defined types and a programming language that does not. 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 is no need to break it down into, say, a sequence of number pairs representing the (*x,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!).

Tutorial D has been designed to support all of the prescriptions and proscriptions of *The Third Manifesto* as defined in Chapter 4. It deliberately does not support all of the "very strong suggestions" mentioned in that chapter, though it does support some of them (possible extensions to deal with others are considered briefly in Chapter 10). The language is also deliberately not 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, as should already be clear from Chapter 2.) However, it is at least true that the shorthands in question are specifically designed to be shorthands [31]; i.e., the redundancies are deliberate, and are included for usability reasons.

Most of the rest of this chapter consists of a BNF grammar for **Tutorial D**. The grammar is defined by means of what is 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 appearing on the left side of some BNF production rule). Then:

- The expression <*xyz list*> denotes a sequence of zero or more <*xyz*>s in which each pair of adjacent <*xyz*>s is separated by at least one space.
- The expression <*xyz commalist*> denotes a sequence of zero or more <*xyz*>s in which each pair of adjacent <*xyz*>s is separated by a comma (as well as, optionally, one or more spaces 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 contained commalist of <*assign*>s is empty degenerates to a <*no op*> ("no operation"). Occasionally, however, there is something a little more interesting

[3] Actually the term *impedance mismatch* is used to mean several different things, of which the mismatch referred to here, between database and language types, is only one.

to be said about such cases (see Exercise 9 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 rules to the contrary. The category <*identifier*> in turn is terminal and is not defined 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——i.e., natural language——explanation of the syntactic category being defined (or one 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 and have not already been given in earlier chapters.[4] (For this reason among others, the grammar is not suitable for driving a mechanical parser, nor is it meant to be; instead, it is meant to serve as an understandable, albeit fairly formal, definition of the constructs that are syntactically valid in the language. A grammar suitable for mechanical parsing can be found at the website *www.thethirdmanifesto.com*.)
- Please note that braces "{" and "}" in the grammar stand for themselves; i.e., they are symbols in the language being defined, not symbols of the metalanguage as they usually are. To be more specific, we use braces to enclose commalists of items when the commalist in question is intended to denote a set of some kind, implying that (a) the order in which the items appear within that commalist is immaterial and (b) if an item appears more than once, it is treated as if it appeared just once (usually; the exceptions are EXACTLY and the *n*-adic versions of COUNT, SUM, AVG, and D_UNION, *q.v.,* for which repeated items have significance). Note, therefore, that if for some <*xyz*> *A* and *B* are <*xyz commalist*>s enclosed in braces that differ only in the order in which the individual <*xyz*>s appear, then *A* and *B* denote the same thing and are regarded as interchangeable.
- The language defined by this 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 does not denote a value; instead, it causes some action to occur, such

[4] For a formal definition of the semantics of the relational algebra operators in particular, however, see Appendix A.

as assigning a value to some variable or changing the flow of control.

- As already noted, various extensions to the language as defined by this grammar are proposed later in the book (especially in Part IV). A syntactic summary of the entire language——including the inheritance extensions from Part IV, but excluding extensions motivated merely by certain of the suggestions in Chapter 10——can be found in Appendix I.
- Finally, the language defined by this grammar constitutes a significant revision of **Tutorial D** as defined in this book's predecessor (reference [83]). For readers who might be familiar with the earlier version, the section "Recent Language Changes" (following the sections on the grammar *per se*) summarizes the most important of those revisions.

COMMON CONSTRUCTS

```
<type>
    ::=   <scalar type>
        | <tuple type>
        | <relation type>

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

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

<relation type>
    ::=   <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 { <parameter name commalist> } ;
             <statement>
          END OPERATOR
```

The *<parameter def commalist>* is enclosed in parentheses instead of braces, as is the corresponding *<argument 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] The <*parameter name commalist*> identifies parameters that are subject to update.

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 Part IV). Analogous remarks apply to <*user read-only op def*>s as well, *q.v.*

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

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

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

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

<argument>
    ::=   <exp>

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

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

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

<tuple exp>
```

[5] Observe that this remark is true of read-only as well as update operators. In particular, it is true of scalar selector operators——that is, the arguments to a <*scalar selector inv*> are specified as a commalist in parentheses, even though the corresponding parameters are specified as a commalist in braces (see <*possrep def*>, later; see also the section "A Remark on Syntax" at the end of the chapter).

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

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

    <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, in *SWE*. The individual <*name intro*>s in *NIC* are executed in sequence as written. As the next production rule shows, each such <*name intro*> includes an <*exp*> and an <*introduced name*>. Let *NI* be one of those <*name intro*>s, and let the <*exp*> and the <*introduced name*> in *NI* be *X* and *N,* 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. Analogous remarks apply to <*tuple with exp*>s and <*relation with exp*>s, *q.v.*

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

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

    <relation with exp>
        ::=   WITH <name intro commalist> : <relation exp>

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

    <selector inv>
        ::=   <scalar selector inv>
            | <tuple selector inv>
            | <relation selector inv>

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

    <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>
```

The semantics of <*assignment*> are those of *multiple* assignment, as required and specified by RM Prescription 21.

```
<assign>
    ::=   <scalar 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 | CHAR | BOOLEAN
```

As indicated, **Tutorial D** supports the following built-in scalar types:

- INTEGER (signed integers): literals expressed as an optionally signed decimal integer; usual arithmetic and comparison operators, with usual notation.
- RATIONAL (signed rational numbers): 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: 5., 5.0, 17.5, -5.3E+2); usual arithmetic and comparison operators, with usual notation.
- CHARACTER or CHAR (varying-length character strings): 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. By the way, if you are familiar with SQL, do not 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). **Tutorial D** does not support a fixed-length character string type.
- BOOLEAN (truth values): literals TRUE and FALSE; usual comparison operators (= and ≠) and boolean operators (AND, OR, NOT, etc.), with usual notation. 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 XOR operator (exclusive OR). The expression *a* XOR *b* (where *a* and *b* are <*bool exp*>s) is semantically identical to the expression *a* ≠ *b*.
  2. It supports *n*-adic versions of the operators AND, OR, and XOR. The syntax is:

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

The <*n-adic bool op name*> is AND, OR, 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. 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*.[6] Then the overall expression evaluates to TRUE if and only if the number of specified <*bool exp*>s that evaluate to TRUE is *N*. *Note:* If the number of specified <*bool exp*>s is zero——i.e., if the <*bool exp commalist*> is empty——the comma following the <*integer exp*> must be omitted.

In practice we would expect a variety of other built-in 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 discussed in Part IV. All types are root types in the absence of inheritance support.

<*user scalar root type def*>
    ::= TYPE <*user scalar type name*> [ORDINAL]
        <*possrep def list*>

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

<*possrep component def*>
    ::= <*possrep component name*> <*type*>

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

[6] The detailed syntax of <*integer exp*>s is not specified in this chapter; however, we note that an <*integer exp*> is of course a numeric expression and hence a <*scalar exp*> also.

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>

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

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

If <*scalar type*> and the INIT specification both appear, <*scalar exp*> must be of type <*scalar type*>. If <*scalar type*> appears, the scalar variable is of that type; otherwise it is of the same type as <*scalar exp*>. If the INIT specification appears, the scalar variable is initialized to the value of <*scalar exp*>; otherwise it is initialized to an implementation-defined value.

TUPLE DEFINITIONS

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

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

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

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

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

If <*tuple type*> and the INIT specification both appear, <*tuple exp*> must be of type <*tuple type*>. If <*tuple type*> appears, the tuple variable is of that type; otherwise it is of the same type as <*tuple exp*>. If the INIT specification appears, the tuple variable is initialized to the value of <*tuple exp*>; otherwise it is initialized to an implementation-defined value.

RELATIONAL DEFINITIONS

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

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

A <*relation var def*> defines a relation variable (i.e., a relvar). In practice it might be desirable to provide a way of defining relation constants or "relcons" also (see RM Prescription 14 in Chapter 6 for further discussion). Note that **Tutorial D** already supports two built-in "relcons" called TABLE_DEE and TABLE_DUM (see the section "Relational Operations," later).

```
<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 is 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 <relation type or init value>
                   <candidate key def list>
```

The keyword REAL can alternatively be spelled BASE. An empty <*candidate key def list*> is permitted, though not required, only if (a) the <*relation type or init value*> specifies or includes INIT (<*relation exp*>) or (b) <*relation type*> is of the form SAME_TYPE_AS (<*relation exp*>); it is equivalent to a <*candidate key def list*> that contains exactly one <*candidate key def*> for each key that can be inferred by the system from the <*relation exp*> in that INIT or SAME_TYPE_AS specification (see RM Very Strong Suggestion 3 in Chapter 10).

```
<relation type or init value>
    ::=   <relation type> | INIT ( <relation exp> )
        | <relation type> 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*> and the INIT specification both appear, <*relation exp*> must be of type <*relation type*>. If <*relation type*> appears, the relvar is of that type; otherwise it is of the same type as <*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 is initialized to the empty relation of the appropriate type.

```
<candidate key def>
    ::=   KEY { <attribute ref commalist> }
```

In accordance with the discussions in Chapter 2, we use the unqualified keyword KEY to mean a candidate key specifically. **Tutorial D** does not explicitly support primary keys as such.

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

The <*relation exp*> must mention at least one database relvar and no other variables. An empty <*candidate key def list*> is equivalent to a <*candidate key def list*> that contains exactly one <*candidate key def*> for each key that can be inferred by the system from <*relation exp*> (see RM Very Strong Suggestion 3 in Chapter 10).

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

An empty <*candidate key def list*> is permitted, though not required, only if (a) the <*relation type or init value*> specifies or includes INIT (<*relation exp*>) or (b) <*relation type*> is of the form SAME_TYPE_AS (<*relation exp*>); it is equivalent to a <*candidate key def list*> that contains exactly one <*candidate key def*> for each key that can be inferred by the system from the <*relation exp*> in that INIT or SAME_TYPE_AS specification (see RM Very Strong Suggestion 3 in Chapter 10).

```
<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 mention any variable that is not a database relvar. (**Tutorial D** does not support the definition of constraints on scalar variables or tuple variables or application relvars, though there is no logical reason why it should not do so.)

```
<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>

<built-in scalar op inv>
    ::=   <scalar selector inv>
        | <THE_ op inv>
        | <attribute extractor inv>
        | <agg op inv>
        | ... plus the usual possibilities
```

It is 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 have already said are built-in 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:

- A special form of <*bool exp*>, IS_EMPTY (<*relation exp*>), which returns TRUE if and only if the relation denoted by <*relation exp*> is empty. (In practice, it might be useful to support an IS_NOT_EMPTY operator as well.)
- 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-THEN-ELSE and CASE expressions of the usual form (and we assume without going into details that tuple and relation analogs of these expressions are available also).

The syntax of <*scalar selector inv*> has already been explained (see Chapter 3 for several examples). *Note:* Whether scalar selectors are regarded as built-in 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*>
[, <*attribute ref*>])

The <*integer exp*> and following comma must be specified if and only if the <*agg op name*> is EXACTLY. The <*attribute ref*> must be omitted if the <*agg op name*> is COUNT; otherwise, it can be omitted if and only if the <*relation exp*> denotes a relation of degree one, in which case the sole attribute of that relation is assumed by default. For SUM and AVG, the attribute denoted by <*attribute ref*> must be of some type for which the operator "+" is defined; for MAX and MIN, it must be of some ordinal type; for AND, OR, XOR, and EXACTLY, it must be of type BOOLEAN; for UNION, D_UNION, and INTERSECT, it must be of 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*> (potentially), or a <*relation exp*> depending on the type of the operator denoted by <*agg op name*>. (Given the aggregate operators currently defined, for UNION, D_UNION, and INTERSECT it is a <*relation exp*>, otherwise it is a <*scalar exp*>.)

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

COUNT returns a result of type INTEGER; SUM, AVG, MAX, MIN, UNION, D_UNION, and INTERSECT return a result of the same type as the attribute denoted by the applicable <*attribute ref*>;[7] AND, OR, XOR, and EXACTLY return a result of type BOOLEAN. The <*agg op name*>s AND and OR can alternatively be spelled ALL and ANY, respectively. *Note:* **Tutorial D** includes support for *n*-adic versions of (a) AND, OR, XOR, and EXACTLY (see the section "Scalar Definitions," earlier) and (b) UNION, D_UNION, INTERSECT, and JOIN (see the section "Relational Operations," later). It also includes support for *n*-adic versions of COUNT, SUM, AVG, MAX, and MIN; for example, SUM {1,2,5,2} is a valid <*scalar exp*>, and it evaluates to 10.

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

<*scalar target*>

[7] 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.

::= <*scalar var ref*>
| <*scalar THE_ pv ref*>

The abbreviation *pv* stands for *pseudovariable*. Pseudovariables are regarded as variables in **Tutorial D,** implying among other things that a pseudovariable reference can appear wherever the grammar requires a variable reference. *Note:* As mentioned in Chapter 3, it would be possible, if desired, to include support for other kinds of pseudovariables in addition to the THE_ pseudovariables mentioned in this grammar. In particular, it would be possible to support pseudovariables patterned after **Tutorial D**'s existing attribute extractors.

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

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

<*scalar update*>
::= UPDATE <*scalar target*>
(<*possrep component assign commalist*>)

Let the <*scalar target*>, *ST* say, be of type *T*. Every <*possrep component assign*>, *PCA* say, in the <*possrep component assign commalist*> is syntactically identical to an <*assign*>, except that:

- The target of *PCA* must be a <*possrep component target*>, *PCT* say.
- *PCT* must identify, directly or indirectly,[8] 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 in Chapter 4 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*

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

for some *Ci,* and no two distinct <*possrep component assign*>s identify 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*>

<*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 to ordinal types only.

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*>

Although we generally have little to say regarding operator precedence, we find it convenient to give high precedence to tuple projection in particular. An analogous remark applies to relational projection also (see later).

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

<tuple component>
    ::=   <attribute ref> <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*>.

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

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

The <*tuple exp*>s must be such that if the tuples denoted by any two of those <*tuple exp*>s have any attributes in common, then the corresponding attribute values are the same.

```
<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 substitute>

<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 executed in sequence as written.

```
<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*> ADD (<*extend add commalist*>)

The <*tuple exp*> must not be a <*monadic or dyadic other built-in tuple op inv*>. The individual <*extend add*>s are executed in sequence as written.

<*extend add*>
::= <*exp*> AS <*introduced name*>

Both <*tuple extend*> and <*extend*> make use of <*extend add*>. We explain both cases here, but it is convenient to treat them separately:

- In the <*tuple extend*> case, the <*exp*> is allowed to include an <*attribute ref*>, *AR* say, wherever a <*selector inv*> would be allowed. If the <*attribute name*> of *AR* is that of an attribute of the tuple denoted by the <*tuple exp*> in that <*tuple extend*>, then it denotes the corresponding attribute value; otherwise the <*tuple extend*> must be contained in some expression in which the meaning of *AR* is defined.
- In the <*extend*> case, the <*exp*> is again allowed to include an <*attribute ref*>, *AR* say, wherever a <*selector inv*> would be allowed. Let *r* be the relation denoted by the <*relation exp*> in that <*extend*>. The <*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; otherwise the <*extend*> must contained in some expression in which the meaning of *AR* is defined.

<*tuple wrap*>
::= <*tuple exp*> WRAP (<*wrapping commalist*>)

The <*tuple exp*> must not be a <*monadic or dyadic other built-in tuple op inv*>. The individual <*wrapping*>s are executed in sequence as written.

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

<*tuple unwrap*>
::= <*tuple exp*> UNWRAP (<*unwrapping commalist*>)

The <*tuple exp*> must not be a <*monadic or dyadic other built-in tuple op inv*>. The individual <*unwrapping*>s are executed in sequence as written.

<*unwrapping*>
    ::= <*attribute ref*>

The specified attribute must be of some tuple type.

<*tuple substitute*>
    ::= UPDATE <*tuple exp*> (<*attribute assign commalist*>)

Syntactically, a <*tuple substitute*> is identical to a <*tuple update*>, except that it contains a <*tuple exp*> in place of the <*tuple target*> required in a <*tuple update*>. 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*>, 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 *t;* otherwise the <*tuple substitute*> must be contained in 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 in Chapter 4 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 identify the same target *Ai*. Now consider the expression

UPDATE *t* (*Ai* := *X*, *Aj* := *Y*)

where *i* ≠ *j*. (For definiteness, we consider the case where there are exactly two <*attribute assign*>s; the revisions needed to deal with other cases are straightforward.) This expression is equivalent to the following:

((EXTEND *t* ADD (*X* AS *Bi*, *Y* AS *Bj*)) { ALL BUT *Ai*, *Aj* })
                        RENAME (*Bi* AS *Bk*, *Bj* AS *Aj*, *Bk* AS *Ai*)

Here *Bi, Bj,* and *Bk* are arbitrary distinct names that are different from all existing attribute names in *t*.

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

<*attribute THE_ pv ref*>

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

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

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

The <*dyadic tuple union*> *r* UNION *s* is equivalent to the <*n-adic tuple union*> UNION {*r,s*}.

<*tuple compose*>
::= <*tuple exp*> COMPOSE <*tuple exp*>

The <*tuple exp*>s must not be <*monadic or dyadic other built-in tuple op inv*>s. They must be such that if the tuples they denote have any attributes in common, then the corresponding attribute values are the same.

<*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 <*possrep component*> corresponding to <*THE_ pv name*> must be of 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 in 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 in Chapter 4 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 identify the same target *Ai*. Now consider the <*tuple update*>

UPDATE *TT* (*Ai* := *X*, *Aj* := *Y*)

where $i \neq j$. (For definiteness, we consider the case where there are exactly two <*attribute assign*>s; the revisions needed to deal with other cases are straightforward.) This <*tuple update*> is equivalent to the following <*tuple assign*>:

TT := UPDATE *TT* (*Ai* := *X*, *Aj* := *Y*)

(The expression on the right side here is a <*tuple substitute*> invocation.)

<*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 "∈" ("epsilon") denotes the set membership operator; it can be pronounced "belongs to" or "is a member of" or just "[is] in." The expression $t \notin r$ is defined to be semantically equivalent to the expression NOT($t \in r$).

<*tuple comp op*>
::= = | ≠

RELATIONAL OPERATIONS

<*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 explicitly, (a) <*heading*> must be specified if <*tuple exp commalist*> is empty; (b) every <*tuple exp*> in <*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 the <*relation selector inv*>s RELATION{}{TUPLE{}} and RELATION{}{}, respectively (see RM Prescription 10 in Chapter 6 for further explanation).

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

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

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

<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 <*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*> and <*n-adic intersect*>, *q.v.*

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

The relations denoted by the <*relation exp*>s must be pairwise disjoint.

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

If the <*relation exp commalist*> is empty, the <*n-adic intersect*> evaluates to the "universal" relation of the applicable type: i.e., the unique relation of that type that contains all possible tuples with the applicable <*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> }

<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> | <substitute> | <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 executed in sequence as written.

```
<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; 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>
                 ADD ( <extend add commalist> )
```

The <*relation exp*> must not be a <*monadic or dyadic other built-in relation op inv*>. The individual <*extend add*>s are executed in sequence as written.

```
<wrap>
    ::=   <relation exp> WRAP ( <wrapping commalist> )
```

The <*relation exp*> must not be a <*monadic or dyadic other built-in relation op inv*>. The individual <*wrapping*>s are executed in sequence as written.

```
<unwrap>
    ::=   <relation exp> UNWRAP ( <unwrapping commalist> )
```

The <*relation exp*> must not be a <*monadic or dyadic other built-in relation op inv*>. The individual <*unwrapping*>s are executed in sequence as written.

```
<group>
    ::=   <relation exp> GROUP ( <grouping commalist> )
```

The <*relation exp*> must not be a <*monadic or dyadic other built-in relation op inv*>. The individual <*grouping*>s are executed in sequence as written.


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

<ungroup>
    ::=    <relation exp> UNGROUP ( <ungrouping commalist> )
```

The *<relation exp>* must not be a *<monadic or dyadic other built-in relation op inv>*. The individual *<ungrouping>*s are executed in sequence as written.

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

The specified attribute must be of some relation type.

```
<substitute>
    ::=   UPDATE <relation exp>
                ( <attribute assign commalist> )
```

Syntactically, a *<substitute>* is identical to a *<relation update>*, except that it contains a *<relation exp>* in place of the *<relation target>* (and optional WHERE *<bool exp>*) required in a *<relation update>*. 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>*, 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 *<substitute>* must be contained in 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 in Chapter 4 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 identify the same target *Ai*. Now consider the expression

UPDATE *r* (*Ai* := *X*, *Aj* := *Y*)

where $i \neq j$. (For definiteness, we consider the case where there are exactly two *<attribute assign>*s; the revisions needed to deal

with other cases are straightforward.) This expression is equivalent to the following:

```
( ( EXTEND r ADD ( X AS Bi, Y AS Bj ) ) { ALL BUT Ai, Aj } )
                          RENAME ( Bi AS Bk, Bj AS Aj, Bk AS Ai )
```

Here *Bi, Bj,* and *Bk* are arbitrary distinct names that are different from all existing attribute names in *r*.

<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 attributes of that relation must both be of the same type.

<dyadic other built-in relation op inv>
::= *<dyadic union>* | *<dyadic disjoint union>*
| *<dyadic intersect>* | *<minus>* | *<dyadic join>*
| *<compose>* | *<semijoin>* | *<semiminus>*
| *<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 relations denoted by the *<relation exp>*s must be disjoint.

<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.

<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>*.

<compose>
    ::= *<relation exp>* COMPOSE *<relation exp>*

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

<semijoin>
    ::= *<relation exp>* SEMIJOIN *<relation exp>*

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

<semiminus>
    ::= *<relation exp>* SEMIMINUS *<relation exp>*

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

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

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

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

Reference [34] 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. See RM Prescription 18 in Chapter 6 for further explanation.

<summarize>
    ::= SUMMARIZE *<relation exp>* [*<per or by>*]
                ADD (*<summarize add commalist>*)

The *<relation exp>* must not be a *<monadic or dyadic other built-in relation op inv>*. Omitting *<per or by>* is equivalent to specifying PER (TABLE_DEE). The individual *<summarize add>*s are executed in sequence as written.

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

Let *r* be the relation to be summarized. If *<per>* is specified, it must contain exactly one *<relation exp>*. Let *pr* be the relation denoted by that *<relation exp>*. Then every attribute of *pr* must be an attribute of *r*. Specifying BY {*A1,A2,...,An*} is equivalent to specifying PER (*r*{*A1,A2,...,An*}).

<summarize add>
    ::= *<summary>* AS *<introduced name>*

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

Let *r* and *pr* 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 or EXACTLYD. The <*integer exp*> is allowed to include an <*attribute ref*>, *IAR* say, wherever a <*selector inv*> would be allowed. If the <*attribute name*> of *IAR* is that of an attribute of *pr,* then *IAR* denotes the corresponding attribute value from some tuple of *pr;* otherwise the <*summary*> must be contained in some expression in which *IAR* is defined.
- The <*scalar exp*> must be specified if and only if the <*summary spec*> is not COUNT. The <*scalar exp*> is allowed to include an <*attribute ref*>, *SAR* say, wherever a <*selector inv*> would be allowed. If the <*attribute name*> of *SAR* is that of an attribute of *r,* then *SAR* denotes the corresponding attribute value from some tuple of *r;* otherwise the <*summary*> must be contained in some expression in which *SAR* is defined.

For SUM, SUMD, AVG, and AVGD, the value denoted by <*scalar exp*> must be of some type for which the operator "+" is defined; for MAX and MIN, it must be of some ordinal type; for AND, OR, XOR, EXACTLY, and EXACTLYD, it must be of type BOOLEAN; for UNION, D_UNION, and INTERSECT, it must be of some relation type. Observe that <*summary*> and <*agg op inv*> are not the same thing, although the type of any given <*summary*> is the same as that of its <*agg op inv*> counterpart.

```
<summary spec>
    ::=   COUNT | COUNTD | SUM | SUMD | AVG | AVGD | MAX | MIN
        | AND | OR | XOR | EXACTLY | EXACTLYD
        | UNION | D_UNION | INTERSECT
```

The suffix "D" ("distinct") in COUNTD, SUMD, AVGD, and EXACTLYD means "eliminate redundant duplicate values before performing the summarization." COUNT and COUNTD return a result of type INTEGER; SUM, SUMD, AVG, AVGD, MAX, MIN, UNION, D_UNION, and INTERSECT return a result of the same type as the value denoted by the applicable <*scalar exp*>;[9] AND, OR, XOR, EXACTLY, and EXACTLYD return a result of type BOOLEAN. The <*summary spec*>s AND and OR can alternatively be spelled ALL and ANY, respectively.

[9] It might be preferable in practice to define the <*summary spec*>s AVG and AVGD 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 assign>
    ::=   <relation target> := <relation exp>
        | <relation insert>
        | <relation delete>
        | <relation update>

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

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

The *<possrep component>* corresponding to *<THE_ pv name>* must be of some relation type. *Note:* Let *rx* be the *<relation exp>* appearing in the *<virtual relation var def>* that defines some virtual relvar *V*. Then it would be possible, assuming *V* is updatable (see Appendix E), 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 delete>
    ::=   DELETE <relation target> [ WHERE <bool exp> ]
```

Let the *<relation target>* be *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 an attribute of *RT,* then (for each such evaluation) *AR* denotes the corresponding attribute value from the corresponding tuple; otherwise the *<relation delete>* must be contained in some expression in which the meaning of *AR* is defined.

```
<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 in 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 in 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 in Chapter 4 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 identify the same target *Ai*. Now consider the <*relation update*>

UPDATE *RT* WHERE *b* (*Ai* := *X*, *Aj* := *Y*)

where *i* ≠ *j*. (For definiteness, we consider the case where there are exactly two <*attribute assign*>s; the revisions needed to deal with other cases are straightforward.) This <*relation update*> is equivalent to the following <*relation assign*>:

```
RT := ( RT WHERE NOT ( b ) )
      UNION
      ( UPDATE RT WHERE b ( Ai := X, Aj := Y ) )
```

The third line here consists of a <*substitute*> invocation in parentheses.

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

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

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

Note: 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 prohibits tuple-at-a-time retrieval from a relation as supported by, e.g., FETCH via a cursor in SQL. 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-at-a-time retrieval can be obtained, if desired, by first performing such a mapping and then iterating over the resulting array.[10] 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 do not define a syntactic category called <*array type*>.

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

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. If it contains at least one, the lower bound is one, otherwise it and the upper bound are both zero. 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. 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 we do not specify any such mechanisms 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.

[10] In accordance with RM Proscription 7, **Tutorial D** supports nothing at all analogous to SQL's tuple-at-a-time *update* operators (i.e., UPDATE or DELETE "WHERE CURRENT OF *cursor*").

- The headings associated with <*array target*> and <*relation exp*> would normally have to 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 were loaded into the array variable to be defined in terms of attributes whose values were not 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.
- LOAD is really *assignment,* of a kind (in particular, it has the effect of replacing whatever value the target previously had). However, we deliberately do not use assignment syntax for it because it effectively involves an implicit type conversion (i.e., a *coercion*) between a relation and an array. We have already given our reasons in Chapter 3 for not wishing to support coercions; 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*>

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 exp*> and an <*array cardinality*> operator (a special case of <*integer exp*>):

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

<*subscript*>
::= <*integer exp*>

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

STATEMENTS

<*statement*>
::= <*statement body*> ;


```
<statement body>
    ::=   <previously defined statement body>
        | <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 execution of the current transaction and to begin a new ("child") transaction (see OO Prescription 5 in Chapter 8 for further explanation). COMMIT or ROLLBACK terminates execution of the transaction most recently begun, thereby reinstating as current——and continuing execution of——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> ]
```

The <*exp*> is required for a read-only operator and prohibited for an update operator. *Note:* An update operator need not contain a <*return*> at all, in which case an implicit <*return*> is executed when the END OPERATOR is reached.

```
<case>
    ::=   CASE ;
             <when def list>
```

```
            [ ELSE <statement> ]
          END CASE

<when def>
    ::=   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>
    ::=   ... an empty string

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

One final point to close this section: Elsewhere in this book, 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 inside itself; i.e., it is not a <*case*>, <*if*>, <*do*>, <*while*>, or compound statement.

RECENT LANGUAGE CHANGES

There are a number of differences between **Tutorial D** as described in the present chapter and the version of the language defined in this book's predecessor (reference [83]). For the benefit of readers who might be familiar with that earlier version, we summarize the main differences here.

- The previous version allowed certain braces or parentheses to be omitted if what was contained within those braces or

parentheses consisted of just one item (or sometimes no item at all). The present version does not.

- In many places the previous version required some list or commalist to be nonempty where the present version does not.
- The previous version allowed possrep component names to be omitted, but the present version does not.
- The ability has been added (a) to define a tuple variable to have the same heading as a specified relation expression and (a) to define a relvar to have the same heading as a specified tuple expression.
- Support for the boolean operators XOR and EXACTLY has been added. For AND, OR, and XOR, both infix (dyadic) and prefix (*n*-adic) syntax are supported (of course, EXACTLY is intrinsically *n*-adic).
- Ordinal types are now explicitly declared as such.
- Update operators are no longer regarded as (or limited to being) scalar; thus, scalar, tuple, and relation parameters can all be subject to update and identified as such in the UPDATES specification. Update operators, but not read-only operators, can also directly update variables that are not local to the operator in question.
- The commalist of <*assign*>s in UPDATE (various forms) is now enclosed in parentheses instead of braces.
- BASE has been introduced as an alternative spelling for REAL.
- The keywords LOCAL and GLOBAL on <*application relation var def*>s have been replaced by PRIVATE and PUBLIC, respectively.
- An INIT specification is now supported for REAL (or BASE) and PRIVATE relvars.
- INIT specifications can now be used to determine the type of the variable being declared.
- The initializing expression in INIT is now enclosed in parentheses, as is the defining expression in a virtual relvar definition.
- A new form of <*scalar assign*> has been added, using the keyword UPDATE.
- For syntactic reasons, tuple join has been replaced by tuple union (semantically, of course, the operators are equivalent).
- Support for disjoint union (D_UNION) has been added.
- Prefix (*n*-adic) versions of union (including tuple union), disjoint union, intersect, and join are now supported.

- MATCHING and NOT MATCHING have been introduced as alternative spellings for SEMIJOIN and SEMIMINUS, respectively.
- The operators <*substitute*>, <*tuple substitute*>, and <*tuple compose*> have been introduced.
- A BY form of SUMMARIZE has been added.
- AND and OR have been introduced as preferred spellings for ALL and ANY, respectively. COUNT is now written COUNT (). Aggregate operators XOR, EXACTLY, UNION, D_UNION, and INTERSECT have been introduced. All of the aggregate operators have both (a) *n*-adic forms and (b) "summary" analogs in SUMMARIZE. Also, the aggregate operators COUNT, SUM, AVG, and EXACTLY have additional "summary" analogs COUNTD, SUMD, AVGD, and EXACTLYD, for which redundant duplicate values are eliminated before the summarization is done.
- The IN operator is now written ∈.
- GROUP and UNGROUP now support "multiple" grouping and ungrouping.
- The syntax of the ordering specification on <*relation get*> has changed.
- The <*with*> statement has been dropped and WITH expressions have been clarified.
- A number of minor corrections have been made.

In addition to all of 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.

A REMARK ON SYNTAX

You might have noticed that the syntax of operator invocations in **Tutorial D** is not very consistent. To be specific:

- User-defined operators use a prefix style, with positional argument/parameter matching.
- Built-in operators, by contrast, sometimes use an infix style ("+", "=", MINUS, etc.), sometimes a prefix style (MAX, EXACTLY, *n*-adic JOIN, etc.).
- Some of those built-in operators rely on positional argument/parameter matching ("+", MINUS, MAX, EXACTLY, etc.), while others do not[11] ("=", *n*-adic JOIN, etc.). Also, those that rely on positional matching use parentheses to enclose their arguments, while those that do not use braces.

[11] Or, at least, the order in which the arguments are specified in such cases is immaterial.

- 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 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) should not 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 in earlier sections of this chapter because we did not 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 5.0 }
```

as a possible replacement for

```
CARTESIAN ( 5.0, 2.5 )
```

(note in particular that the parentheses have been replaced by braces). In other words, the suggestion is that a general <*op inv*> ("operator invocation") 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*> <*exp*>

There are some difficulties, however. For one thing, this new prefix style seems clumsier than the old in the common special case in which the operator takes just one parameter, as with (e.g.) SIN, COS, and sometimes COUNT. For another, some common operators (e.g., "+", "=", ":=") have names that do not abide by the usual rules for forming identifiers. For a third, built-in 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 , ... , P49 r49 }
```


Again, however, the new syntax in this particular case seems clumsier than the old, 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 is the P1 parameter and which the P2 (*r1* MINUS *r2* and *r2* MINUS *r1* are not equivalent, in general). Some additional apparatus would be required to communicate such information to the user.

EXERCISES

1. Write a set of **Tutorial D** data definitions for the suppliers-and-parts database (relvar definitions only; Exercise 14 in Chapter 3 already asked for the type definitions).
2. Define virtual relvars for (a) suppliers with status greater than ten; (b) shipments of red parts; (c) parts not available from any London supplier.
3. Distinguish between database and application relvars.
4. Is the boolean operator XOR associative?
5. Consider the prefix (*n*-adic) versions of AND, OR, and XOR. What happens if the specified <*bool exp commalist*> contains just one <*bool exp*>? What if it contains none at all?
6. The expression XOR {<*bool exp commalist*>} is defined to evaluate to TRUE if and only if an odd number of the specified <*bool exp*>s evaluate to TRUE. Justify this definition.
7. What does the expression EXACTLY (0,{<*bool exp commalist*>}) return? What if the <*bool exp commalist*> is empty?
8. Give **Tutorial D** formulations for the following updates to the suppliers-and-parts database:
   a. Insert a new shipment with supplier number S1, part number P1, quantity 500.
   b. Insert a new supplier S10 (name and city Smith and New York, respectively; status not yet known).
   c. Delete all blue parts.
   d. Delete all parts for which there are no shipments.
   e. Change the color of all red parts to orange.
   f. Replace all appearances of supplier number S1 by appearances of supplier number S9 instead.

In each case, give two formulations, one using INSERT, DELETE, or UPDATE (as applicable) and one using a pure relational assignment.

9. The **Tutorial D** grammar presented in this chapter involves numerous lists and commalists. In every case, what happens if the list or commalist is empty?

10. The LOAD statement involves an ORDER specification. Considered as an operator in its own right, however, ORDER is rather unusual. In what respects?

11. Consider the following type definition:

```
TYPE ELLIPSE POSSREP { A RATIONAL, B RATIONAL, CTR POINT
                       CONSTRAINT A ≥ B } ;
```

(This is a simplified version of an example we will be using extensively in later chapters.) Now let E be a variable of type ELLIPSE, and consider the following two statements:

a. `THE_A ( E ) := 7.0 , THE_B ( E ) := 5.0 ;`

b. `UPDATE E ( A := 7.0 , B := 5.0 ) ;`

Is there any logical difference between these statements? If so, what is it?

12. Data definition operations (for objects in the database, at least) cause updates to be made to the catalog. But the catalog is only a collection of relvars, just like the rest of the database; so could we not just use the familiar update operations INSERT, DELETE, and UPDATE to update the catalog appropriately? Discuss.

13. Design an extension to the syntax of the **Tutorial D** SUMMARIZE operator that would make, e.g., an expression of the form

```
( MAX ( X ) - MIN ( Y ) ) / 2
```

a valid <*summary*>.

14. Suppose we are given the following definitions:

```
TYPE POINT POSSREP { X RATIONAL, Y RATIONAL } ;
TYPE CIRCLE POSSREP { R RATIONAL, CTR POINT } ;
TYPE COLORED_CIRCLE POSSREP { CIR CIRCLE, COL COLOR } ;

VAR CC COLORED_CIRCLE ;
```

The following is a valid assignment statement with variable CC as target:

```
UPDATE CC ( UPDATE CIR ( UPDATE CTR ( X := 2 * X ) ) ) ;
```

Show a completely expanded version of this statement that makes no use of the UPDATE shorthand.

Date and Darwen, DATABASES TYPES & THE RELATIONAL MODEL, pp. 93-125,

***** End of Chapter 5 *****

