# 2 Solutions (shown in blue)

## 2.1 Exercise for Chapter 1, Introduction

Consider the following table (from Figure 1.5 of the book)

| A | B | A |
|---|---|---|
| 1 | 2 | 3 |
| 4 | | 5 |
| 6 | 7 | 8 |
| 9 | 9 | ? |
| 1 | 2 | 3 |

Give three reasons why it cannot possibly represent a relation.

1. Two of the columns have the same name, A. The attributes of a relation have unique names for identification purposes.
2. The first and last of the rows shown in green are identical. The same tuple cannot appear more than once in the body of a relation.
3. The second row shown in green appears to have no entry for column B. Every tuple in the body of a relation has exactly one value for each attribute of that relation's heading.

Some students might think there is a fourth error, concerning the question mark in the last column of the penultimate row. Each attribute of a relation has a defined type, and each tuple in that relation must have for that attribute a value of its type. If the values shown in the third column are all of the same type, then it is a type that contains a value that can be denoted by the symbol "?" as well as several values denoted by numbers. That is perhaps an improbable type but relational theory places no restrictions as to which types are permissible as attribute types.

## 2.2 Exercises for Chapter 2, Values, Types, Variables, Operators

Complete sentences 1–10 below, choosing your fillings from the following:

=, :=, ::=, argument, arguments, body, bodies, `BOOLEAN`, cardinality, `CHAR`, `CID`, degree, denoted, expressions, false, heading, headings, `INTEGER`, list, lists, literal, literals, operator, operators, parameter, parameters, read-only, set, sets, `SID`, true, type, types, update, variable, variables.

In 1–5, consider the expression `X = 1 OR Y = 2`.

1. In the given expression, = and `OR` are _____ whereas `X` and `Y` are _____ references.

   operators, variable

2. `X` and `1` denote _____ to an invocation of _____.

   arguments, =

3. The value _____ by the given expression is of _____ `BOOLEAN`.

   denoted, type

4. `1` and `2` are both _____ of _____ `INTEGER`.

   literals, type

5. The operators used in the given expression are _____ operators.

   read-only

In 6–10, consider the expression `RELATION { X SID, Y CID } { }`.

6. It denotes a relation whose _____ is zero and whose _____ is two.

   cardinality, degree *Explanation:* the cardinality is the number of tuples in the body and the degree is the number of attributes in the heading.

7. It is a relation _____.

   literal

8. The declared type of `Y` is _____.

   `CID`

9. In general, the heading of a relation is a possibly empty _____ of attributes and its body is a possibly empty _____ of tuples.

   set, set

10. It is _____ that the assignment RV __ `RELATION { X SID, Y CID } { }` is legal if the _____ of RV is `{ Y CID, X SID }`, _____ that it is legal if the _______ of RV is `{ A SID, B CID }`, _____ that it is legal if the _____ of RV is `{ X CID, Y SID }`, and _____ that it is legal if the _____ of RV is `{ X CHAR, Y CHAR }`.

true, :=, heading, false, heading, false, heading, false, heading

Solutions to questions posed in exercises in **Getting Started with *Rel***

5. Why do we have to write `output x ;` in full when it is part of a compound statement, instead of just `x`?

Because otherwise *Rel* might be looking at `x end ;` and that is not a valid statement of any kind. The presence of a line break carries no significance.

What have you learned about *Rel's* rules concerning *case sensitivity*?

Identifiers are case-sensitive, key words are not.

6. When "Enhanced" is off, is the output of evaluating the given relation literal identical to the input?

> No. The output includes `{CourseId CHAR, Name CHAR, StudentId CHAR}` in between the key word `RELATION` and the first opening brace. Also, the character string literals are enclosed in double-quotes instead of single-quotes.

Now delete all the tuple expressions, leaving just `RELATION { }`. What happens when *Rel* tries to evaluate that?

> You get an error message saying that "{" is expected in place of <EOF>. In other words, it expects another list enclosed in braces to follow the empty one.

Now use < to recall the original `RELATION` expression to the input pane and re-evaluate it with "Enhanced" *off*. Use copy-and-paste to copy the result to the input pane, then delete all the `TUPLE` expressions, to leave this:

```
RELATION {StudentId CHARACTER, CourseId CHARACTER,
          Name CHARACTER} { }
```

Study the result of that in the output pane, first with "Enhanced" off, then with it on.

What conclusions do you draw from all this, about *Rel* and **Tutorial D**?

> The text inserted after the key word `RELATION` can be recognized as a specification of the *heading* of the relation: a list of the attribute names and their declared types (in this example, `CHARACTER` for each attribute). **Tutorial D** allows the heading to be specified, in which case each tuple specified in the *body* must be of that heading. **Tutorial D** also allows the heading to be omitted, *provided that the body is not empty*. Each tuple must of course be of the same heading, and that determines the heading of the relation.

> *Rel* allows `CHAR` literals to be enclosed in either single-quotes or double-quotes. The closing quote must match the opening one.

Next, enter the following literal, perhaps by using the < button to recall `enrolment` and editing it:

```
RELATION {
TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' },
TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' }
}
```

Before you press Evaluate (F5), think about what you expect to happen. Does the result meet your expectation? How do you explain it?

> The body of a relation is a *set* of tuples. A set by definition contains exactly one appearance of each of its elements. *Rel* would perhaps be justified in treating this expression as an error, but it is equally justified in just ignoring any duplicate tuples. In conventional mathematical notation, {1,2,3,1}, for example, is considered to denote the set consisting of the elements 1, 2, and 3. The redundancy can sometimes be convenient when variables are involved—the set {*x*, *y*}, for example, has cardinality 1 in the case where *x*=*y*.

Use < again to recall the `enrolment` literal. Insert `WITH ( enrolment :=` at the beginning and add `) : enrolment` at the end, to give:

> The `WITH` expression equates the name `enrolment` with the `RELATION` expression preceding the key word `AS`. It is the expression following the colon (:) that *Rel* evaluates. So in this simple case, `WITH` defines the name `enrolment`, and `enrolment` is then the expression we ask *Rel* to evaluate when we click on Run (F5).

By inspection of `enrolment` only, write down all the cases you can find of two students such that there is at least one course they are both enrolled on.

> Anne and Boris
> Boris and Devinder
> Anne and Devinder

If you included all the cases where the two students are in fact the same student, such as "Anne and Anne", well, that's a good point—the question didn't say "*distinct* students".

7. How many distinct projections can be obtained from `enrolment`?

> Eight. If you found less than eight, did you forget the empty projection, `enrolment{}`? If you found more than eight, were you perhaps thinking that, for example, `enrolment{StudentId, Name}` and `enrolment{Name, StudentId}` are distinct projections? Recall that attribute order carries no significance.

8. Your renaming should look like this:

```
WITH ( enrolment :=
RELATION {
 TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne'   },
 TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne'   },
 TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
 TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
 TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
} , E1 := enrolment RENAME
                    { StudentId AS SID1, Name AS N1 } ) :
E1
```

9. Here is the expression you should have evaluated:

```
WITH ( enrolment :=
RELATION {
 TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne'   },
 TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne'   },
 TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
 TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
 TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
} , E1 := enrolment RENAME
                     {StudentId AS SID1, Name AS N1 },
 E2 := enrolment RENAME
                     { StudentId AS SID2, Name AS N2 }
 ) :
E1 JOIN E2
```

How do you interpret the result? How many tuples does it contain? Replace the key word `JOIN` by `COMPOSE`. How do you interpret *this* result? How many tuples are there now? How do you account for the difference?

The result of the join gives pairs of students, shown by their names and ids, enrolled on the same course, along with the course id of that course. There are 11 tuples, including several in which the two students are in fact the same person!

The result of the compose gives pairs of students such that there is at least one course they are both enrolled on. This time there are only 10 tuples, because Anne is enrolled on two courses and therefore appears twice, paired with herself, in the join but only once in the composition. (The composition is equivalent to the join followed by `{ ALL BUT CourseId }`).

10. Add `WHERE NOT ( SID1 = SID2 )` to end of the expression you evaluated in Step 9. Examine the result closely. Now place parentheses around `E1 COMPOSE E2` and evaluate again. Confirm that you get the same result.

Repeat the experiment, replacing `WHERE NOT ( SID1 = SID2 )` by `{ SID1 }`. Do you get the same results this time? If not, why not?

What does all this tell you about operator precedence rules in *Rel*?

Because presence of parentheses around `e1 COMPOSE e2` makes no difference when that is followed by an invocation of `WHERE`, it appears that `COMPOSE` takes precedence over restriction. And so does `JOIN`. However, when we replace the restriction by a projection that specifies an attribute of `E1`, we find that it fails unless the `COMPOSE` invocation is enclosed in parentheses. We conclude that projection takes precedence over `COMPOSE` (and `JOIN`). *On the whole you are left to discover Rel's operator precedence rules for yourself. Of course you can always use parentheses to override them, as in most computer languages.*

Why was it probably a good idea to add that `WHERE` invocation? Does it completely solve the problem? If not, can you think of a better solution?

It eliminates the cases of the two students paired together being the very same student. However, we are still left with Anne being paired with Boris in one tuple, and Boris being paired with Anne in another tuple. Obviously if Anne and Boris are both enrolled on some course, we don't really want to be told so twice. It seems that the relation for the predicate '$x$ is enrolled on the same course as $y$' is *reflexive* (true whenever $x = y$) and symmetric (if it is true when $x = a$ and $y = b$, then it is true when $x = b$ and $y = a$).

We can eliminate the redundant cases by using the `WHERE` condition `SID1 < SID2`, sneakily taking advantage of the fact that character strings are ordered (in **Rel**, as in most programming languages, of course).

What connection, if any, do you see between this exercise and Exercise 6?

See the last paragraph of Exercise 6.

## 2.3 Exercises for Chapter 3, Predicates and Propositions

Consider again the relation shown as the current value of ENROLMENT in Figure 1.2:

| StudentId | Name | CourseId |
|---|---|---|
| S1 | Anne | C1 |
| S1 | Anne | C2 |
| S2 | Boris | C1 |
| S3 | Cindy | C3 |
| S4 | Devinder | C1 |

For each of the following propositions, state whether it is true or false, basing your conclusions on this relation:

1. There exists a course *CourseId* such that some student named Anne is enrolled on *CourseId*.

   **True**—C1 is such a course.

2. Every student with StudentId S1 who is enrolled on some course is named Anne.

   **True**—we might guess that the students named Anne who are enrolled on both C1 and C2 are in fact the same student, but we are not actually told that. Even if for some strange reason they are different people, they are both named Anne and no other enrolment is for somebody with StudentId S1.

3. Every student who is enrolled on course C4 is named Anne.

**True**—there does not exist a student who is enrolled on C4 and is not named Anne. Recall that "for all *x*, *P*(*x*)" is logically equivalent to "there does not exist *x* such that NOT ( *P*(*x*) )".

4. Some student who is enrolled on course C4 is named Anne.

**False**—as nobody at all is enrolled on C4 it cannot be possible for anybody named Anne to be enrolled on it.

5. There are exactly 5 students who are enrolled on some course.

**False**—there are 4. However, this relies on the assumption that no two students have the same StudentId, in which case those two S1's are indeed the same student; so the answer "can't tell" is perhaps even more acceptable.

6. It is not the case that there is no course on which no student who is enrolled on some course but is not named Boris is not enrolled.

**False**—Cindy is not enrolled on C1; Cindy and Devinder are not enrolled on C2; Anne and Devinder are not enrolled on C3. If course C4 exists, then nobody is enrolled on it, so Anne, Cindy and Devinder are each not enrolled on it. So for each course there is at least one student who is not enrolled on it but is enrolled on some course and is not named Boris. So it *is* the case there is no course having no such student.

7. There are exactly 10 pairs of StudentIds (*SID1, SID2*) such that there is some course on which student *SID1* is enrolled and student *SID2* is enrolled.

**True**—the pairs in question are (Anne, Boris), (Anne, Devinder), (Boris, Devinder), (Devinder, Boris), (Devinder, Anne), (Boris, Anne), (Anne, Anne), (Boris, Boris), (Devinder, Devinder), and (Cindy, Cindy).

8. There are exactly 3 pairs of StudentIds (*SID1, SID2*) such that there is some course on which student *SID1* is enrolled and student *SID2* is enrolled.

**False**—we have already shown that there are 10.

9. If a student named Eve is enrolled on course C1, then student S1 is named Adam.

**True**—because "a student named Eve is enrolled on course C1" is false. Recall that a proposition of the form "If *p*, then *q*" is defined to be **False** only when *p* is **True** and *q* is **False**. So, whenever *p* is **False**, "If *p*, then *q*" is **True**.

10. If student S1 is named Anne, then S1 is enrolled on course C2.

**True**—because "S1 is named Anne" and "S1 is enrolled on course C1" are both true.

## 2.4 Exercises for Chapter 4, Relational Algebra – The Foundation

1. Recall that *r1* `TIMES` *r2* requires *r1* and *r2* to have no common attributes, in which case it is equivalent to *r1* `JOIN` *r2*. Why would it be a bad idea to *require* `TIMES` to be used in place of `JOIN` in such cases?

Consider relations R1 { A, B }, R2 { B, C }, and R3 { C, D }. We have seen that, thanks to the commutativity and associativity of JOIN, we can join these three together in any order. For example: ( R1 JOIN R3 ) JOIN R2. But if we are required to use TIMES instead of JOIN for the join of R1 and R3, that particular expression is illegal.

2. Given the following relvars:

```
VAR Cust BASE RELATION {C# CHAR, Discount RATIONAL} KEY {C#};
VAR Orders BASE RELATION {O# CHAR, C# CHAR, Date DATE}
                     KEY {O#};
VAR OrderItem BASE RELATION {O# CHAR, P# CHAR, Qty INTEGER }
                        KEY {O#, P#};
VAR Product BASE RELATION {P# CHAR, Unit_price RATIONAL}
                      KEY {P#};
```

The price of an order item can be calculated by the formula:

```
CAST_AS_RATIONAL(Qty)*Unit_price*(1.0-(Discount/100.0))
```

Write down a relation expression to yield a relation with attributes `O#`, `P#`, and `Price`, giving the price of each order item.

```
WITH ( COI := Cust JOIN Orders JOIN OrderItem JOIN Product ) :
EXTEND COI : { Price := CAST_AS_RATIONAL(Qty)*Unit_price*
                        (1.0-(Discount/100.0))}
       { O#, P#, Price }
```

3. Given:

```
VAR Exam_Marks BASE RELATION { StudentId SID,
                               CourseId CID,
                               Mark INTEGER}
                         KEY { StudentId, CourseId };
```

Write down a relational expression to give, for each pair of students sitting the same exam, the absolute value of the difference between their marks. Assume you can write `ABS(x)` to obtain the absolute value of *x*.

```
WITH ( EM1 := EXAM_MARK RENAME { StudentId AS S1, Mark as M1 },
       EM2 := EXAM_MARK RENAME { StudentId AS S2, Mark as M2 },
       EM1_2 := EM1 JOIN EM2,
       Sat_same_exam := EM1_2 WHERE S1 <> S2 ) :
EXTEND Sat_same_exam : { Diff := ABS ( M1–M2 ) )
{ S1, S2, Diff }
```

4. State the value of

(a) *r* `NOT MATCHING TABLE_DEE`

The empty relation with the heading of *r* (every tuple matches the 0-tuple)

(b) *r* `NOT MATCHING TABLE_DUM`

*r* (`DUM` has no tuples for the tuples of *r* to match with)

(c) *r* `NOT MATCHING` *r*

The empty relation with the heading of *r* (every tuple in *r* matches some tuple in *r*, namely, itself)

(d) (*r* `NOT MATCHING` *r* ) `NOT MATCHING` *r*

The empty relation with the heading of *r* (if the first operand of `NOT MATCHING` is empty, then so is the result)

(e) *r* `NOT MATCHING` (*r* `NOT MATCHING` *r*)

*r* (if the second operand of `NOT MATCHING` is empty, then the result is the first operand)

Is `NOT MATCHING` associative? Is it commutative?

Examples (d) and (e) above show that it is not associative. It is not commutative because the heading of the result is that of the first operand.

5. (Repeated from the body of the chapter) Which operator, in the list given in Section 4.11, **Concluding Remarks**, can be dispensed with without sacrificing relational completeness? How can it be defined in terms of the other operators?

`RENAME` is redundant. *r* `RENAME { ` ***a*** `AS` ***b*** ` }` is equivalent to

```
( EXTEND r : { b := a } ) { ALL BUT a }
```

6. (Repeated from the body of the chapter) Investigate the completeness of an algebra that includes `MINUS` in place of `NOT MATCHING` by attempting to define `NOT MATCHING` in terms of `MINUS` and the other operators.

*r1* `NOT MATCHING` *r2* is equivalent to

```
r1 JOIN ( r1 { c } MINUS r2 { c } )
```

where *c* is a commalist of the names of the attributes common to *r1* and *r2*. Therefore an algebra that contains the listed operators but with `MINUS` in place of `NOT MATCHING` is indeed relationally complete. Note that no harm is done to the given expression is *c* happens to be empty.

7. The chapter briefly mentions the operator `MATCHING` but defers its detailed description to Chapter 5. Before you read that chapter, define *r1* `MATCHING` *r2* in terms of the operators described in Chapter 4.

*r1* `MATCHING` *r2* , which yields the relation consisting of those tuples of r1 that match some tuple in *r2,* is equivalent to

```
r1 JOIN ( r1 { c } JOIN r2 { c } )
```

where *c* is a commalist of the names of the attributes common to *r1* and *r2*. It is also equivalent to

```
( r1 JOIN r2 ) { hr1 }
```

where *hr1* is a commalist of the names of the attributes of *r1*.

**Working with a Database in *Rel***

1. Start up *Rel*.
2. Figure 4.13 shows the supplier-and-parts database from Chris Date's *Introduction to Database Systems (8th edition)*, as shown on the inside back cover of that book (except that the attribute names there are in upper case).

**S**

| S# | Sname | Status | City |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

**SP**

| S# | P# | Qty |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S1 | P3 | 400 |
| S1 | P4 | 200 |
| S1 | P5 | 100 |
| S1 | P6 | 100 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S4 | P4 | 300 |
| S4 | P5 | 400 |

**P**

| P# | Pname | Color | Weight | City |
|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London |
| P2 | Bolt | Green | 17.0 | Paris |
| P3 | Screw | Blue | 17.0 | Oslo |
| P4 | Screw | Red | 14.0 | London |
| P5 | Cam | Blue | 12.0 | Paris |
| P6 | Cog | Red | 19.0 | London |

**Figure 4.13:** The suppliers-and-parts database

Execute a **Tutorial D** VAR statement for each of S, P and SP. Use INTEGER as the declared type for STATUS and QTY, RATIONAL for WEIGHT, and CHAR for all the other attributes. Feel free to use lower case or mixed case to suit your own taste for attribute and relvar names, but do not otherwise change any of the given names.

**Tutorial D** requires at least one key constraint to be specified for each relvar. One key for each for S, P and SP is shown by underlining the relevant attribute names in the table. No other key constraints are needed.

```
VAR S BASE RELATION { S# CHAR, Sname CHAR,
                      Status INTEGER, City CHAR}
          KEY { S# } ;

VAR P BASE RELATION { P# CHAR, Pname CHAR,
                      Colour CHAR, Weight RATIONAL,
                      City CHAR}
        KEY { P# } ;

VAR SP BASE RELATION {S# CHAR, P# CHAR, Qty INTEGER}
        KEY { S#, P# } ;
```

"Populate" (as they say) each relvar with the values shown in Date's tables. There are several ways of achieving this. Choose whichever you prefer from the following:

a. Include an `INIT ( ... )` specification in the `VAR` statement, after the heading and before the `KEY` specification. Inside the parens, write a `RELATION { ... }` expression, using a `TUPLE` expression for each required tuple, as in the enrolment literal used in the *Rel* exercises for Chapter 2.

```
VAR S BASE RELATION { S# CHAR, Sname CHAR,
                      Status INTEGER, City CHAR}
          KEY { S# }
          INIT ( RELATION {
                   TUPLE { S# 'S1', Sname 'Smith',
                           City 'London',
                           Status 20 },
                   TUPLE { S# 'S2', etc. } }
               ) ;
```

b. Execute the `VAR` statement without an `INIT ( ... )` specification. The implied initial value is the empty relation of the appropriate type. You can see this by asking *Rel* for the current value of the relvar. For example, to get the current value of `S`, just type `S` into the lower pane and click `Run (F5)`.
Now use an assignment statement of the form
*`varname := relation-expression`*
to populate the relvar. Check that *Rel* has indeed assigned the correct value to it.

```
S := RELATION {
     TUPLE { S# 'S1', Sname 'Smith',
             City 'London', Status 20 },
     TUPLE { S# 'S2', etc. } } ;
```

c. Use *Rel* `INSERT` statements to populate the relvar piecemeal, perhaps one tuple at a time. Having typed in the first `INSERT` statement. Here is the general form of an `INSERT` statement to insert a single tuple:

```
INSERT varname RELATION { TUPLE { ... } } ;
```

Note that the source operand is still a relation, not just a tuple, hence the need to enclose the `TUPLE` expression inside `RELATION { }`.

```
INSERT S RELATION {
            TUPLE { S# 'S1', Sname 'Smith',
                    City 'London', Status 20 } } ;
```

and repeat for each tuple to be inserted.

3. Informally, we refer to `S` as suppliers, `P` as parts and `SP` as shipments. Predicates for these relvars are:

**`S`:** Supplier *S#* is named *Sname*, has status *Status* and is located in city *City*.

**`P`:** Part *P#* is named *Pname*, is coloured *Color*, weighs *Weight* and is located in city *City*.

**`SP`:** Supplier *S#* ships part *P#* in quantities of *Qty*.

What, then, is the predicate for the expression `S JOIN SP JOIN P`?

Supplier *S#* is named *Sname*, has status *Status* and is located in city *City* and part *P#* is named *Pname*, is coloured *Colour*, weighs *Weight* and is located in city *City* and Supplier *S#* ships part *P#* in quantities of *Qty*.

What do you expect to be the result of that expression?

*Rel* gives this (with Enhanced not checked):

```
RELATION {S# CHAR, Sname CHAR, Status INTEGER, City CHAR, P#
CHAR, Qty INTEGER, Pname CHAR, Colour CHAR, Weight RATIONAL}
{ TUPLE {S# "S1", Sname "Smith", Status 20, City "London",
P# "P1", Qty 300, Pname "Nut", Colour "Red", Weight 12.0},
TUPLE {S# "S1", Sname "Smith", Status 20, City "London",
P# "P4", Qty 200, Pname "Screw", Colour "Red", Weight 14.0},
TUPLE {S# "S4", Sname "Clark", Status 20, City "London",
P# "P4", Qty 300, Pname "Screw", Colour "Red", Weight 14.0},
TUPLE {S# "S1", Sname "Smith", Status 20, City "London",
P# "P6", Qty 100, Pname "Cog", Colour "Red", Weight 19.0},
TUPLE {S# "S3", Sname "Blake", Status 30, City "Paris",
P# "P2", Qty 200, Pname "Bolt", Colour "Green", Weight 17.0},
TUPLE {S# "S2", Sname "Jones", Status 10, City "Paris",
P# "P2", Qty 400, Pname "Bolt", Colour "Green", Weight 17.0}
}
```

In tabular form (i.e., with Enhanced checked):

| S# | Sname | Status | City | P# | Pname | Colour | Weight | Qty |
|---|---|---|---|---|---|---|---|---|
| S1 | Smith | 20 | London | P1 | Nut | Red | 12.0 | 300 |
| S1 | Smith | 20 | London | P4 | Screw | Red | 14.0 | 200 |
| S2 | Jones | 10 | Paris | P2 | Bolt | Green | 17.0 | 400 |
| S3 | Blake | 30 | Paris | P2 | Bolt | Green | 17.0 | 200 |
| S4 | Clark | 20 | London | P4 | Screw | Red | 14.0 | 300 |
| S1 | Smith | 20 | London | P6 | Cog | Red | 19.0 | 100 |

What is its degree? 9 (the number of attributes)

Does *Rel* give the result you expected? Explain what you see.

`S` and `P` both have an attribute named `City`, so this is a common attribute for matching purposes, as well as `S#` and `P#`. Note the two appearances of `City` in the predicate for the join. They must both stand for the same city.

4. Attempt to insert a tuple into SP with supplier number S1, part number P1 and quantity 100. Explain the result of your attempt.

*Rel* gives this:

```
INSERT SP RELATION { TUPLE { S# 'S1', P# 'P1', Qty 200 } } ;
ERROR: Inserting tuple would violate uniqueness constraint
of KEY {S#, P#}
Line 1, column 62 near '100'
```

The declaration of relvar SP includes the specification KEY { S#, P# }, which means the same as, for example:

```
CONSTRAINT SPkey COUNT(SP{S#,P#})=COUNT(SP);
```

In other words, the cardinality of SP must always be the same as that of its projection over S# and P#. In other words, for any given combination of S# and P# values, there must be at most one tuple in SP. Successful insertion of TUPLE { S# 'S1', P# 'P1', Qty 200 } would have resulted in SP containing two tuples with S# = 'S1' and P# = 'P1', thus violating the constraint.

5. Get *Rel* to evaluate each of the following expressions. For each one, write down the corresponding predicate and also give an informal interpretation of the query in the style of those given in Exercise 6 below.

a. `SP WHERE P# = 'P2'`

Supplier *S#* ships part *P#* in quantities of *Qty* and *P#* is P2.

*Note that although we have fixed the value for P#, we haven't actually substituted P2 for P# in the predicate!*

Get shipments of part P2.

b. `S { ALL BUT Status }`

There exists a status *Status* such that supplier *S#* is named *Sname*, has status *Status* and is located in city *City*.

Get all information about suppliers, apart from their status.

c. `SP { S#, Qty }`

There exists a part number *P#* such that Supplier *S#* ships part *P#* in quantities of *Qty*.

For each supplier, get the various quantities used for shipments.

d. `P NOT MATCHING ( SP WHERE S# = 'S2' )`

Supplier *S#* is named *Sname*, has status *Status* and is located in city *City* and there exists a quantity *Qty* such that *S#* ships part P2 in quantities of *Qty*.

*In these predicates the parameters (free variables) are shown in bold to distinguish them from the bound variable Qty. Qty is bound by use of the quantifier, "there exists". You can use the more formal mathematical notation for existential quantification if you prefer:*

∃ *Qty* (Supplier *S#* is named *Sname*, has status *Status* and is located in city *City* and *S#* ships part P2 in quantities of *Qty*)

Get suppliers who supply part P2.

e. `S MATCHING ( SP WHERE P# = 'P2' )`

Part *P#* is named *Pname*, is coloured *Colour*, weighs *Weight* and is located in city *City* and there does not exist a quantity *Qty* such that supplier S2 ships *P#* in quantities of *Qty*.

Get parts that supplier S2 cannot supply.

f. `S { City } UNION P { City }`

There exist a supplier number *S#*, a name *Sname*, and a status *Status* such that Supplier *S#* is named *Sname*, has status *Status* and is located in city *City*, or there exist a part number *P#*, a name *Pname*, a colour *Colour*, and a weight *Weight* such that part *P#* is named *Pname*, is coloured *Colour*, weighs *Weight* and is located in city *City*.

or, if you prefer,

∃ *S#* ∃ *Sname* ∃ *Status* (supplier *S#* is named *Sname*, has status *Status* and is located in city *City*)—∃ *P#* ∃ *Pname* ∃ *Colour* ∃ *Weight* (part *P#* is named *Pname*, is coloured *Colour*, weighs *Weight* and is located in city *City*)

Note the use of—to signify disjunction ("or").

Get cities where either a supplier or a part is located.

g. `S { City } MINUS P { City }`

There exist a supplier number *S#*, a name *Sname*, and a status *Status* such that Supplier *S#* is named *Sname*, has status *Status* and is located in city *City*, and there do not exist a part number *P#*, a name *Pname*, a colour *Colour*, and a weight *Weight* such that part *P#* is named *Pname*, is coloured *Colour*, weighs *Weight* and is located in city *City*.

Get cities where a supplier is located but no part is located.

h.
```
( ( S RENAME { City AS SC } ) { SC } ) JOIN
( ( P RENAME { City AS PC } ) { PC } )
```

There exist a city *City*, a name *Sname*, a status *Status*, a name *Pname*, a colour *Colour*, and a weight *Weight* such that Supplier *S#* is named *Sname*, has status *Status* and is located in city *City*, and part *P#* is named *Pname*, is coloured *Colour*, weighs *Weight* and is located in city *City*.

*Sometimes it seems impossible to write an informal interpretation without having recourse to the kind of variable symbols we use in predicates. Hence:*

Get <*S#*, *P#*> pairs such that supplier *S#* and part *P#* are located in the same city.

6. Write **Tutorial D** expressions for the following queries and get *Rel* to evaluate them:

a. Get all shipments.

```
SP
```

b. Get supplier numbers for suppliers who supply part P1.

```
S MATCHING ( SP WHERE P# = 'P1' ) { S# }
```

c. Get suppliers with status in the range 15 to 25 inclusive.

```
S WHERE Status > 14 AND Status < 26
```

d. Get part numbers for parts supplied by a supplier in Paris.

```
( SP JOIN ( S WHERE City = 'Paris' ) ) { P# }
```

e. Get part numbers for parts not supplied by any supplier in Paris.

```
P { P# } NOT MATCHING
  ( SP JOIN ( S WHERE City = 'Paris' ) )
```

f. Get city names for cities in which at least two suppliers are located.

```
( ( S {S#, City} RENAME{S# AS S#1}
 JOIN
 S {S#, City} RENAME{S# AS S#2} )
 WHERE S#1 < S#2 ) {City}
```

or you can use `SUMMARIZE`:

```
( ( SUMMARIZE S PER ( S { City } ) :
                    { No_of_Supps := COUNT() } )
WHERE No_of_Supps > 1 ) { City }
```

g. Get all pairs of part numbers such that some supplier supplies both of the indicated parts.

```
( ( SP { S#, P# } RENAME { P# AS Px } )
 JOIN
 ( SP { S#, P# } RENAME { P# AS Py } )
) { Px, Py } WHERE Px < Py
```

The final restriction is optional. It assumes that only pairs of distinct part numbers are required, and that we do not want the result to include both `TUPLE { PX` *`px`*`, PY` *`py`* `}` and `TUPLE { PX` *`py`*`, PY` *`px`* `}` for any (*px*, *py*).

h. Get supplier numbers for suppliers with a status lower than that of supplier S1.

```
( S JOIN ( ( ( S WHERE S# = 'S1' ) { Status }
             RENAME { Status AS S1_Status }
           )
         )
  WHERE Status < S1_Status
) { S# }
```

i. Get supplier-number/part-number pairs such that the indicated supplier does not supply the indicated part.

```
( S { S# } JOIN P { P# } ) NOT MATCHING SP
```

## 2.5 Exercises for Chapter 5, Building on The Foundation

1. (Repeated from the body of the chapter) What can you say about the result of *r1* `COMPOSE` *r2* when *r1* and *r2* have identical headings? For example, what is the result of `IS_CALLED COMPOSE IS_CALLED`?

   The result is a relation of degree zero: TABLE_DEE if some tuple in *r1* matches some tuple in *r2*, otherwise TABLE_DUM. All attributes are common and common attributes do not appear in the result. When *r1* = *r2*, the result is TABLE_DUM if and only if *r1* is empty.

2. (Repeated from the body of the chapter) Is `COMPOSE` associative? In other words, is ( *r1* `COMPOSE` *r2* ) `COMPOSE` *r3* equivalent to *r1* `COMPOSE` ( *r2* `COMPOSE` *r3* )? If so, prove it; if not, show why.

   It is not associative. For let *r1*, *r2*, and *r3* be defined as follows:

   ```
   r1 = RELATION {TUPLE { X 2 } }
   r2 = RELATION {TUPLE { X 1 } }
   r3 = r2
   ```

   Then *r1* `COMPOSE` *r2* yields `TABLE_DUM` and `TABLE_DUM COMPOSE` *r3* yields the empty relation with heading the heading of *r3*. However, `r2 COMPOSE` *r3* yields `TABLE_DEE` and*r1* `COMPOSE TABLE_DEE` clearly yields *r1*, which is not empty.

3. What can you say about the result of `r1 MATCHING (` *r2* `MATCHING r1 )`?

   The expression is equivalent to *r1* `MATCHING r2`. The result of `(` *r2* `MATCHING` *r1* `)` contains exactly those tuples of *r2* that match some tuple in *r1*. The loss of the unmatched tuples of *r2* has no effect on the final result.

4. (Repeated from the body of the chapter) Does the aggregate operator `AVG` have a basis operator? If so, define it.

   No. However, the average of a list of values $x_1$, …, $x_n$ ($n \geqslant 1$), can be computed iteratively, as opposed to computing their sum and dividing by the number of elements, by letting `T` be the tuple `TUPLE{X 0, N 0}` and then, for $i$ = 1:$n$, replacing `T` by the tuple `TUPLE{X ((X FROM T) +` $x_i$`)/N FROM T), N` $i$`}`. Then the final result is given by `X FROM T`.

In other words, AVG *per se* does not have a basis operator, but an aggregate operator that yields a 2-tuple giving the required average along with the number of elements contributing to that average, does have a basis operator, as defined.

5. Suppose an aggregate operator PRODUCT is defined, with arithmetic multiplication as its basis operator. What is the result of PRODUCT(*r*,*x*) if *r* is empty?

1, because 1 is the identity value under multiplication ($\forall x, x = x*1$).

6. (Repeated from the body of the chapter) Is it *always* the case that the cardinality of an ungrouping is equal to the sum of the cardinalities of the relations being ungrouped on?

No. For let *r1* be

```
RELATION { TUPLE { X RELATION { TUPLE { Y 1 },
                                 TUPLE { Y 2 } } },
           TUPLE { X RELATION { TUPLE { Y 1 } } } }
```

Then the sum of the cardinalities of the X values in *r1* is 2+1=3, whereas the result of *r1* UNGROUP ( X ) is

```
RELATION { TUPLE { Y 1 },
           TUPLE { Y 2 } }
```

whose cardinality is just 2. The tuple TUPLE { Y 1 } appears in both X values but only once (of course!) in the result of the ungrouping.

7. Write **Tutorial D** expressions for the following queries and get *Rel* to evaluate them:

a. Get the total number of parts supplied by supplier S1.

```
COUNT ( SP WHERE S# = 'S1' )
```

But that expression yields a scalar value. To obtain the result in the form of a relation, one could write, for example,

```
RELATION { TUPLE { Parts_from_S1 COUNT ( SP WHERE S# =
'S1' ) } }
```

or

```
SUMMARIZE ( SP WHERE S# = 'S1' )
          PER ( TABLE_DEE ) :
              { Parts_from_S1 := COUNT() }
```

b. Get supplier numbers for suppliers whose city is first in the alphabetic list of such cities.

```
( S WHERE City = MIN ( S, City ) ) { S# }
```

c. Get part numbers for parts supplied by all suppliers in London.

```
WITH ( LS := S WHERE City = 'London' ,
       T := SP RENAME { P# AS X } ) :
( P WHERE (
    ( T WHERE X = P# ) { S# } ) >= ( LS { S# } ) )
{ P# }
```

Note the use of relation comparison (>= is *Rel's* notation for **Tutorial D**'s ⊇, "is a superset of"). Use of `WITH` is optional, of course. You might come up with a different solution, but does it address the possibility of there being no suppliers at all in London?

Alternatively, just using the operators of the algebra:

```
WITH ( LonSupp := S WHERE City = 'London',
       LonSuppWithAllParts := LonSupp{S#} JOIN P,
       PartNotSuppliedByLonSupp :=
                LonSuppWithAllParts NOT MATCHING SP ) :
P{P#} NOT MATCHING PartNotSuppliedByLonSupp
```

The second line pairs every London supplier's supplier number with every part number. The next two lines find (S#,P#) pairs such that London supplier S# doesn't supply part P#. The last line find part numbers that that aren't part numbers of parts not supplied by some London supplier. Unravelling the double negative makes that mean part numbers of parts supplied by every London supplier. Note how the use of relation comparison in the first solution avoids this double negative.

d. Get supplier numbers and names for suppliers who supply all the purple parts.

```
WITH ( t1 := EXTEND S :
             { Parts_Supplied :=
               ( SP JOIN RELATION { TUPLE { S# S#} } )
              {P#}},
       t2 := t1 WHERE Parts_Supplied >=
                ( P WHERE Colour = 'Purple' ){P#} ) :
t2 { S#, Sname }
```

An alternative solution in the style of the alternative given for 7c. is available here too.

e. Get all pairs of supplier numbers, S*x* and S*y* say, such that S*x* and S*y* supply exactly the same set of parts each.

*The following solution, using relation comparison, appeals directly to the exercise's "the same set":*

```
 WITH ( RX := S RENAME { S# AS SX },
        RY := S RENAME { S# AS SY } ) :

 ( ( RX JOIN RY ) WHERE
                   ( ( SP WHERE S# = SX ) { P# } )
                   = ( ( SP WHERE S# = SY ) { P# } )
) { SX, SY } WHERE SX < SY
```

*Note the importance of referencing S and not SP in the definitions of RX and RY. If we reference SP we might miss suppliers who supply no parts at all.*

*The final restriction is optional—see Exercise 6, part g, in the solutions to the exercise for Chapter 4.*

f. Write a truth-valued expression to determine whether all supplier names are unique in `S`.

```
COUNT(S) = COUNT(S{Sname})
```

g. Write a truth-valued expression to determine whether all part numbers appearing in `SP` also appear in `P`.

```
P{P#} ⊇ SP{P#}
```

In *Rel,* of course, you write >= instead of ⊇.

## 2.6 Exercises for Chapter 6, Constraints and Updating

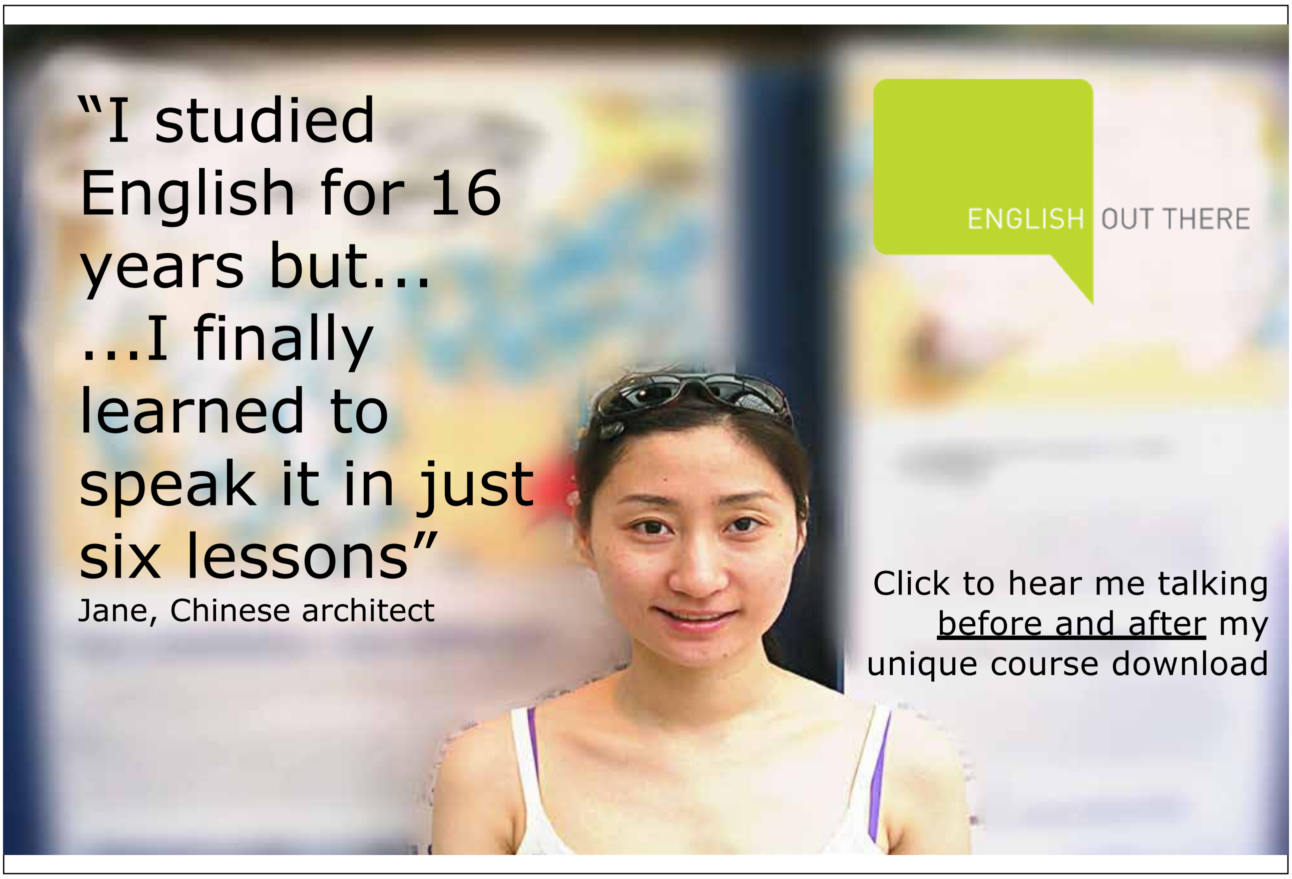
1. (Repeated from the body of the chapter).

a. An implication of `KEY { ALL BUT }` is that no other key can possibly exist for the relvar it applies to. Why is this so?

The specification { ALL BUT } denotes all the attributes—the entire heading—of the relevant relvar. Thus, any additional key must be a proper subset of that heading. But by definition no proper subset of a key is a key (the so-called *irreducibility* property).

b. An implication of `KEY { }` is that no other key can possibly exist for the relvar it applies to. Why is this so?

The specified key is the empty set. Any additional key must be nonempty (there is only one empty set) and therefore a proper superset of the specified key. But a proper superset of a key is by definition not a key (though it *is* a superkey).



2. Suppose the relvar definition for COURSE is extended to include an attribute MaxExamMark, whose value in each tuple is the maximum mark obtainable for that course's exam. {StudentId, CourseId} is a foreign key in EXAM_MARK, referencing IS_ENROLLED_ON. A constraint is needed to ensure that no student is awarded a mark greater than the relevant maximum.

   a. Write a **Tutorial D** CONSTRAINT statement to address this requirement, where the constraint condition is an invocation of IS_EMPTY.

```
CONSTRAINT MarkAcceptable
           IS_EMPTY ( ( EXAM_MARK JOIN COURSE )
                      WHERE Mark > MaxExamMark ) ;
```

   b. Complete the following statement to make it equivalent to the one you wrote for part (a):

```
CONSTRAINT … AND(EXAM_MARK, … ) ;
```

```
CONSTRAINT MarkAcceptable AND ( EXAM_MARK,
      Mark <= ( MaxExamMark FROM TUPLE FROM
                COURSE RENAME { CourseId AS C }
                       WHERE C = CourseId ) );
```

   but the question is a little unfair because of course one would prefer to write

```
CONSTRAINT MarkAcceptable AND(EXAM_MARK JOIN COURSE,
      Mark <= MaxExamMark ) ;
```

3. Now suppose that instead of there being a recorded maximum mark of each exam the maximum score for each question in each exam is recorded in the following relvar:

```
VAR EXAM_QUESTION BASE RELATION { CourseId CID,
      Question# INTEGER, MaxMark INTEGER }
      KEY { CourseId, Question# } ;
```

For each course, the exam questions are supposed to be numbered sequentially, starting at 1.

a. Write a **Tutorial D** `CONSTRAINT` statement to address this requirement.

The constraint is satisfied so long as each question number is between 1 and the number of tuples in `EXAM_QUESTION` for the relevant course:

```
CONSTRAINT QuestionNumbersAcceptable
AND ( EXAM_QUESTION, Question# >= 1 AND Question# <=
         COUNT ( EXAM_QUESTION RENAME { CourseId AS C }
                 WHERE CourseId = C ) ) ;
```

b. Suppose the questions are subdivided into parts, a, b, c and so on, up to a maximum of six parts, and maximum marks are given for each part rather than for each question. Again, the parts for each question must be "numbered" sequentially, starting at a. Write a **Tutorial D** `CONSTRAINT` statement to address *this* requirement.

Assuming the new attribute is named Part, I use a join with a relation literal to map the letters a, b, c, d, e, f to 1, 2, 3, 4, 5, 6, respectively:

```
CONSTRAINT PartNumbersAcceptable
WITH ( AN := RELATION { TUPLE { Part 'a', PN 1 },
                        TUPLE { Part 'b', PN 2 },
                        TUPLE { Part 'c', PN 3 },
                        TUPLE { Part 'd', PN 4 },
                        TUPLE { Part 'e', PN 5 },
                        TUPLE { Part 'f', PN 6 } } ) :
AND ( EXAM_QUESTION JOIN AN, PN >= 1 AND PN <=
      COUNT ( EXAM_QUESTION RENAME { CourseId AS C,
                                     Question # AS Q }
              WHERE CourseId = C AND Question# = Q ) ) ;
```

c. Devise shorthands, in the style of **Tutorial D**, for expressing constraints of the kinds found in your solutions to a. and b.

The requirement for monotonically increasing serial "numbers" arises quite commonly. Usually, the attribute(s) in question are members of some key of the relevant relvar, and the numbering is "within" other elements of the same key. For example, in part b. of the present exercise question parts are numbered within question number and course id, by which we mean that in the set of tuples having the same CourseId and Question# values, Part values range from 'a' to the letter corresponding to the cardinality of that set (e.g., 'e' if there are five such tuples). Similarly, questions are numbered within course id and part, meaning that in the set of tuples having the same CourseId and Part values, Question# values range from 1 to the cardinality of that set.

In general, we probably want our shorthand to allow the starting value and increment each to be something other than 1, if desired. The key words `WITHIN`, `FROM` and `BY` suggest themselves for such specifications, and perhaps the operator name `SERIAL` can be used to identify the kind of shorthand we are after. Typically the constraint applies to just a single relvar but we will allow it to be applied to a relational expression, if only to support mappings such as the one used in the solution to part b.

So, here is a suggested syntax for a truth-valued operator named SERIAL, in the style of **Tutorial D**:

```
IN ( <relation exp> )
     SERIAL  <attribute ref>
   [ WITHIN { <attribute ref commalist> } ]
   [    FROM <integer exp> ]
   [      BY <integer exp> ]
```

The WITHIN specification defaults to WITHIN { } and the two <integer exp>s both default to 1. So constraints required for question numbers and part numbers would be expressed as follows:

```
CONSTRAINT QuestionNumbersAcceptable
IN ( EXAM_QUESTION )
     SERIAL Question# WITHIN { CourseId, Part } ;

CONSTRAINT PartNumbersAcceptable
WITH ( AN := RELATION { TUPLE { Part 'a', PN 1 },
                        TUPLE { Part 'b', PN 2 },
                        TUPLE { Part 'c', PN 3 },
                        TUPLE { Part 'd', PN 4 },
                        TUPLE { Part 'e', PN 5 },
                        TUPLE { Part 'f', PN 6 } } ) :
IN ( EXAM_QUESTION JOIN AN )
     SERIAL PN WITHIN { CourseId, Question# } ;
```

Note that the suggested IN syntax might be used for other kinds of constraint shorthands too, for there is always going to be *<relation exp>* to which the desired constraint is to be applied. Moreover, when the constraint applies to a simple relvar, as in QuestionNumbersAcceptable, the constraint could perhaps be specified, minus the IN specification, within the declaration of that relvar, as with KEY constraints in **Tutorial D**.

4. Using *Rel*, with the suppliers-and-parts database set up for the *Rel* exercises given at the end of Chapter 4, write **Tutorial D** integrity constraints to express the following requirements:

a. Every shipment tuple must have a supplier number matching that of some supplier tuple.

```
CONSTRAINT Ca IS_EMPTY ( SP NOT MATCHING S ) ;
```

Relation comparison could alternatively be used:

```
CONSTRAINT Ca ( SP { S# } ) <= ( S { S# } ) ;
```

but note the need to state the matching attribute name explicitly—this might be thought to be an advantage or a disadvantage. The first solution is neat and immune to changes in attribute names, but exposed to the possibility of inappropriately chosen attribute names. The second solution is exposed to the possible change in name of the S# attributes but is immune to all other attribute name changes. A compromise could be:

```
CONSTRAINT Ca IS_EMPTY ( SP { S# } NOT MATCHING S ) ;
```

b. Every shipment tuple must have a part number matching that of some part tuple.

```
CONSTRAINT Cb IS_EMPTY ( SP NOT MATCHING P ) ;
```

c. All London suppliers must have status 20.

```
CONSTRAINT Cc IS_EMPTY
 ( S WHERE City = 'London' AND Status <> 20 ) ;
```

d. No two suppliers can be located in the same city.

Add the following to the declaration of the relvar S:

```
KEY { CITY }
```

or

```
CONSTRAINT Cd
  COUNT ( S { City } ) = COUNT ( S ) ;
```

e. At most one supplier can be located in Athens at any one time.

```
CONSTRAINT Ce
  COUNT ( S WHERE City = 'Athens' ) <= 1 ;
```

f. There must exist at least one London supplier.

```
CONSTRAINT Cf
 COUNT ( S WHERE City = 'London' ) > 0 ;
```

g. The average supplier status must be at least 10.

One is tempted to write something like

```
CONSTRAINT Cg
 AVG ( S, Status ) >= 10.0 ;
```

but AVG is undefined on the empty set. If it is permissible for S to be empty, we could write

```
CONSTRAINT Cg
 AVG (      S { S#, Status }
            UNION
            RELATION { TUPLE { S# 'S1', Status 10 } } ,
            Status ) >= 10.0 ;
```

but not

```
CONSTRAINT Cg

 IS_EMPTY ( S ) OR AVG ( S, Status ) >= 10.0 ;
```

because **Tutorial D** assumes that (a) operands of OR can be evaluated in either order and (b) the system is permitted to evaluate both operands even when it has discovered one of them to be TRUE.

h. Every London supplier must be capable of supplying part P2.

```
CONSTRAINT Ch IS_EMPTY (
  ( S WHERE City = 'London' ) NOT MATCHING
  ( SP WHERE P# = 'P2' ) ) ;
```

## 2.7 Exercises for Chapter 7, Database Design I: Projection-Join Normalization

1. (Repeated from the body of the chapter). The predicate for `WIFE_OF_HENRY_VIII` is "The first name of the *Wife#*-th wife of Henry VIII is *FirstName* and her last name is *LastName* and *Fate* is what happened to her." Write an appropriate predicate for the following expression:

```
WIFE_OF_HENRY_VIII { Wife#, FirstName }
JOIN
WIFE_OF_HENRY_VIII { LastName, Fate }
```

Note that the join is a Cartesian product—there are no common attributes. So the `LastName` and `Fate` values in a given tuple do not necessarily represent the last name and fate of the wife identified by the `Wife#` value. Bearing that in mind, the predicate must be something like this:

> *FirstName* is the first name of wife number *Wife#* and *LastName* and *Fate* are the last name and fate, respectively, of some wife.

But "some wife" is not very precise. To clarify, we need to write

> *FirstName* is the first name of wife number *Wife#* and there exists a wife number *W#* such that *LastName* and *Fate* are the last name and fate, respectively, of wife *W#*.

2. Consider the following declarations:

```
VAR C1_EXAM_MARK BASE
     INIT ( EXAM_MARK WHERE CourseId = CID('C1') )
      KEY { StudentId } ;

CONSTRAINT C1_only
    AND ( C1_EXAM_MARK, CourseId = CID('C1') ) ;
```

(Recall that `AND` is the aggregate operator mentioned in Chapter 5, evaluating to `TRUE` if and only if the given condition evaluates to `TRUE` for every tuple of the given relation.)

a. Explain why `C1_EXAM_MARK` is not in BCNF.

Because `CourseId` necessarily has the same value in every tuple of `C1_EXAM_MARK`, the nontrivial FD { } → { `CourseId` } holds. The determinant of this FD, being the empty set, is a proper subset of the declared key, { StudentId } and is therefore not a superkey. BCNF requires the determinant of every nontrivial FD to be a superkey.

b. Assume that similar relvars are defined for every course, except that this time there are no `CourseId` attributes. Describe how a query could be expressed to give the course identifier and mark for every exam taken by student S1.

```
EXTEND C1_EXAM_MARK WHERE StudentId = 'S1' :
                    { CourseId := 'C1' }
UNION
EXTEND C2_EXAM_MARK WHERE StudentId = 'S1' :
                    { CourseId := 'C2' }

UNION
EXTEND C3_EXAM_MARK WHERE StudentId = 'S1' :
                    { CourseId := 'C3' }
```

… *(and so on, for each course)*

```
{ CourseId, Mark }
```

3. In Section 7.5 of the chapter, under the heading **Functional Dependencies**, the following eight theorems are given concerning FDs.

1. **Reflexivity:** If $B$ is a subset of $A$, then $A \rightarrow B$
2. **Augmentation:** If $A \rightarrow B$, then $A \cup C \rightarrow B \cup C$
3. **Transitivity:** If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$
4. **Self-determination:** $A \rightarrow A$
5. **Decomposition:** If $A \rightarrow B$ and $C$ is a subset of $B$, then $A \rightarrow C$ and $A \rightarrow B - C$
6. **Union:** If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow B \cup C$
7. **Composition:** If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup C \rightarrow B \cup D$
8. **Unification:** If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup (C - B) \rightarrow B \cup D$

Taking the first three as axioms, prove theorems 4 to 8.

**4. Self-determination:** $A \rightarrow A$

This follows immediately from Theorem 1, Reflexivity, as every set is a subset of itself.

**5. Decomposition:** If $A \rightarrow B$ and $C$ is a subset of $B$, then $A \rightarrow C$ and $A \rightarrow B - C$

| | | |
|---|---|---|
| 1. | $C$ is a subset of $B$ | (given) |
| 2. | $B \rightarrow C$ | (1, reflexivity) |
| 3. | $A \rightarrow B$ | (given) |
| 4. | $A \rightarrow C$ | (3,2, transitivity). |
| 5. | $A \rightarrow B - C$ | (3, reflexivity) |

**6. Union:** If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow B \cup C$

| | | |
|---|---|---|
| 1. | $A \rightarrow B$ | (given) |
| 2. | $A \cup A \rightarrow A \cup B$ | (1, augmentation) |
| 3. | $A \rightarrow A \cup B$ | (2, $A \cup A = A$) |
| 4. | $A \rightarrow C$ | (given) |
| 5. | $A \cup B \rightarrow B \cup C$ | (4, augmentation) |
| 6. | $A \rightarrow B \cup C$ | (3, 5, transitivity) |

**7. Composition:** If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup C \rightarrow B \cup D$

| | | |
|---|---|---|
| 1. | $A \rightarrow B$ | (given) |
| 2. | $A \cup C \rightarrow B \cup C$ | (1, augmentation) |
| 3. | $B \rightarrow D$ | (given) |
| 4. | $B \cup C \rightarrow C \cup D$ | (3, augmentation) |
| 5. | $B \cup C \rightarrow D$ | (4, decomposition) |
| 6. | $B \cup B \cup C \rightarrow B \cup D$ | (5, augmentation) |
| 7. | $B \cup C \rightarrow B \cup D$ | (6, simplification) |
| 8. | $A \cup C \rightarrow B \cup D$ | (2, 7, transitivity) |

**8. Unification:** If $A \to B$ and $C \to D$, then $A \cup (C - B) \to B \cup D$

| | | |
|---|---|---|
| 1. | $A \to B$ | (given) |
| 2. | $C \to D$ | (given) |
| 3. | $A \to B \cap C$ | (1, decomposition) |
| 4. | $C - B \to C - B$ | (self-determination) |
| 5. | $A \cup (C - B) \to (B \cap C) \cup (C - B)$ | (3, augmentation) |
| 6. | $A \cup (C - B) \to C$ | (5, simplification) |
| 7. | $A \cup (C - B) \to D$ | (6, 2, transitivity) |
| 8. | $A \cup (C - B) \to B \cup D$ | (6, 7, union) |

4. (Repeated from the body of the chapter). Consider relvar SCDF with attributes S (for student), C (for course), D (for department), and F (for faculty). Assuming that the set {{C} → {D}, {D} → {F}} is a minimal cover for the FDs in SCDF, prove that {S,C} is a key of SCDF.

| | | |
|---|---|---|
| 1. | {C} → {D} | (given) |
| 2. | {S,C} → {S,D} | (1, augmentation) |
| 3. | {D} → {F} | (given) |
| 4. | {C} → {F} | (1, 3, transitivity) |
| 5. | {S,C} → {S,F} | (4, augmentation) |
| 6. | {S,C} → {S,D,F} | (2, 6, union) |
| 7. | {S,C} → {S,C,D,F} | (6, augmentation) |

So {S,C} is a superkey (7).

To prove that {S,C} is a key it remains to show that neither {S} nor {C} is a key.

8. In the given cover {{C} → {D}, {D} → {F}}, S is not a member of any determinant, so we cannot conclude, for example, that {S} → {F}. Therefore {S} is not a superkey.
9. In the given cover {{C} → {D}, {D} → {F}}, S is not a member of any dependant, so we cannot conclude, for example, that {C} → {S}. Therefore {C} is not a superkey.

So {S,C} is a key (8, 9).

5. (Repeated from the body of the chapter). Assume that {{W,X} → {Y,Z}, {Y} → {X}} is a minimal cover for the FDs in relvar WXYZ. Prove that {W,X} and {W,Y} are both keys of WXYZ.

| | | |
|---|---|---|
| 1. | {W,X} → {Y,Z} | (given) |
| 2. | {W,X} → {W,X,Y,Z} | (1, augmentation, simplification) |

So {W,X} is a superkey (2).

| | | |
|---|---|---|
| 3. | {Y} → {X} | (given) |
| 4. | {W,Y} → {X,Y,Z} | (3, 1, unification) |
| 5. | {W,Y} → {W,X,Y,Z} | (4, augmentation, simplification) |

So {W,Y} is a superkey (5).

6. From the given FDs we cannot conclude {W} → {X}, or {X} → {Z}, or {Y} → {W}, for example, from which it follows that none of {W}, {X}, and {Y} is a superkey, from which it follows in turn that each of the superkeys {W,X} and {W,Y} is indeed a key.

6. The heading of relvar R1 consists of attributes named a, b, c, d, e, f, g, and h. The following set of FDs is a cover for those that hold in R1:

FD1: {a,b} → {c}
FD2: {a,b} → {d}
FD3: {b} → {e}
FD4: {c} → {f}
FD5: {g} → {h}
FD6: {d} → {b, e}

a. Describe the single change required to derive an irreducible cover from the given set.

We can delete the attribute e from the dependant of FD6. This is because from FD6 we have {d} → {b}, from FD3 {b} → {e}, from which {d} → {e} follows by transitivity.

b. Describe the single change required to derive a minimal cover from your answer to a.

FD1 and FD2 Applying Theorem 6, Union to FD1 and FD2 we obtain the single FD {a,b} → {c,d}, from which both FD1 and FD2 can be derived by Theorem 5, Decomposition.

c. Explain why R1 is not in Boyce-Codd normal form (BCNF).

BCNF requires the determinant of every nontrivial FD to be a superkey. From the given nontrivial FDs that hold in R1 we can see several whose determinants are not superkeys. For example, consider FD5 {g} → {h}, As neither g nor h appears in any other given FDs, we cannot conclude that {g} is a determinant for any attribute apart from itself and h, so {g} is not a superkey.

d. Decompose R1 into an equivalent set of BCNF relvars. Name your relvars R2, R3, and so on and for each one list its attribute names and state its key(s). For example: R3{c,d,e} KEY{d} KEY{c,e} if you think this relvar with those two keys is part of the solution.

```
R2{g,h} KEY {g}
R3{d,b} KEY {d}
R4{b,e} KEY {b}
R5{c,f} KEY {c}
R6{a,b,g} KEY{a,b,g}
R7{a,c,d} KEY{a,d}
```

Note that the decomposition loses FD2 {a,b} →{d}, so we would need an additional constraint to the effect that {a,b} is a key for R7 JOIN R3.

7. *Exercise not repeated here.*

In the solutions below the VAR and CONSTRAINT statements are shown below relevant business rule(s). Sometimes the same business rule appears more than once, meaning that the statement(s) shown below it only partially address it.

First, using Option 1 as described in Section 8.4:

**BR1:**

```
VAR customer BASE RELATION { Customer# CHAR,
                             Name CHAR,
                             Address CHAR }
                       KEY { Customer# } ;
```

**BR2:**

```
VAR cust_email BASE RELATION { Customer# CHAR,
                               EmailAddr CHAR }
                         KEY { Customer# } ;

CONSTRAINT FK_for_cust_email
           IS_EMPTY ( cust_email NOT MATCHING customer ) ;
```

**BR3:**

```
VAR phone_type BASE RELATION { Phone# CHAR,
                               PhoneType CHAR }
                         KEY { Phone# } ;

CONSTRAINT HorBorM AND ( phone_type,
                         PhoneType = 'home' OR
                         PhoneType = 'business' OR
                         PhoneType = 'mobile' ) ;
```

**BR4:**

```
VAR cust_phone BASE RELATION { Customer# CHAR,
                               Phone# CHAR }
                         KEY { ALL BUT } ;
```

Note that inclusion of a `type` attribute in `cust_phone` would violate BCNF because of the FD `{Phone#}` → `{PhoneType}`

```
CONSTRAINT at_most_one_phone_per_type
     WITH ( CPT := cust_phone JOIN phone_type ) :
      COUNT ( CPT ) =
      COUNT ( CPT { Customer#, PhoneType } ) ;
```

**BR5:**

```
VAR account BASE RELATION { Account# CHAR,
                            Customer# CHAR,
                            AccountType CHAR,
                            DateOpened DATE }
                        KEY { Account# } ;
```

**BR6:**

```
CONSTRAINT FK_for_account
           IS_EMPTY ( account NOT MATCHING customer ) ;
```

**BR13:**

```
VAR debit_card BASE RELATION { Card# CHAR,
                               Account# CHAR,
                               Holder CHAR,
                               Expires DATE }
                          KEY { Card# } ;

CONSTRAINT FK_for_debit_card
           IS_EMPTY ( debit_card NOT MATCHING account ) ;
```

**BR7, BR8, BR9, BR16:**

```
VAR payment_in BASE RELATION { Trans# CHAR,
                               Account# CHAR,
                               Tdate DATE,
                               Ttime TIME,
                               Source CHAR,
                               Amount RATIONAL }
                         KEY { Trans#, Account# } ;

CONSTRAINT FK_for_payment_in
           IS_EMPTY ( payment_in NOT MATCHING account ) ;
```

**BR7, BR8, BR10, BR16:**

```
VAR by_cheque BASE RELATION { Trans# CHAR,
                              Account# CHAR,
                              Cheque# CHAR,
                              Wdate DATE,
                              Pdate DATE,
                              Payee CHAR,
                              Amount RATIONAL }
                        KEY { Trans#, Account# } ;

CONSTRAINT FK_for_by_cheque
           IS_EMPTY ( by_cheque NOT MATCHING account ) ;
```

**BR11:**

```
CONSTRAINT written_before_processed
           IS_EMPTY ( by_cheque WHERE Wdate > Pdate ) ;
```

**BR7, BR8, BR12, BR16:**

```
VAR by_DD BASE RELATION { Trans# CHAR,
                          Account# CHAR,
                          Tdate DATE,
                          Ttime TIME,
                          Payee CHAR,
                          Amount RATIONAL }
                    KEY { Trans#, Account# } ;

CONSTRAINT FK_for_by_DD
           IS_EMPTY ( by_DD NOT MATCHING account ) ;
```

**BR14, BR16:**

```
VAR by_card BASE RELATION { Trans# CHAR,
                            Card# CHAR,
                            Tdate DATE,
                            Ttime TIME,
                            Payee CHAR,
                            Amount RATIONAL }
                        KEY { Card#, Trans# }
                        KEY { Tdate, Ttime, Card# } ;
```

Note that inclusion of an `account#` attribute in `by_card` would violate BCNF because of the FD {`Card#`} → {`Account#`}

```
CONSTRAINT FK_for_by_card
           IS_EMPTY ( by_card NOT MATCHING debit_card ) ;
```

**BR7:**

```
CONSTRAINT trans#_unique_within_account#
    WITH ( bc_dc := by_card JOIN debit_card ) :
     COUNT ( BC_DC ) =
     COUNT ( BC_DC { Trans#, Account# } ) ;
```

**BR15:**

```
CONSTRAINT card_extant
           IS_EMPTY ( ( by_card JOIN debit_card )
                      WHERE Tdate > Expires ) ;
```

**BR17:**

```
CONSTRAINT transaction_is_of_only_one_type
           WITH ( bc_dc := by_card JOIN debit_card ) :
           IS_EMPTY ( payment_in MATCHING
                    ( by_cheque { Trans#, Account# } UNION
                      by_DD { Trans#, Account# } UNION
                      BC_DC { Trans#, Account# } ) ) AND
           IS_EMPTY ( by_cheque MATCHING
                    ( by_DD { Trans#, Account# } UNION
                      BC_DC { Trans#, Account# } ) ) AND
           IS_EMPTY ( by_DD MATCHING BC_DC ) ;
```

**BR18:**

```
CONSTRAINT no_transaction_before_account_open
           WITH ( bc_dc := by_card JOIN debit_card ) :
           AND ( payment_in JOIN account, DateOpened > Tdate )
           AND
           AND ( by_DD JOIN account, DateOpened > Tdate )
           AND
           AND ( BC_DC JOIN account, DateOpened > Tdate ) ;
```

**Now, using Option 2 instead:**

Relvars `customer`, `cust_email`, `phone_type`, `cust_phone`, `account`, and `debit_card` and their associated constraints are all as before.

**BR7, BR8:**

```
VAR transact BASE RELATION { Trans# CHAR,
                             Account# CHAR,
                             Amount RATIONAL }
                       KEY { Trans#, Account# } ;

CONSTRAINT FK_for_transact
           IS_EMPTY ( transact NOT MATCHING account ) ;
```

**BR10, BR16:**

```
VAR by_cheque BASE RELATION { Trans# CHAR,
                              Account# CHAR,
                              Cheque# CHAR,
                              Wdate DATE,
                              Pdate DATE,
                              Payee CHAR }
                        KEY { Trans#, Account# } ;

CONSTRAINT FK_for_by_cheque
           IS_EMPTY ( by_cheque NOT MATCHING transact ) ;
```

**BR11:**

```
CONSTRAINT written_before_processed
           IS_EMPTY ( by_cheque WHERE Wdate > Pdate ) ;
```

**BR9, BR12:**

```
VAR not_by_cheque BASE RELATION { Trans# CHAR,
                                  Account# CHAR,
                                  Tdate DATE,
                                  Ttime TIME }
                              KEY { Trans#, Account# } ;

CONSTRAINT FK_for_not_by_cheque
           IS_EMPTY ( not_by_cheque NOT MATCHING transact ) ;
```

**BR9:**

```
VAR payment_in BASE RELATION { Trans# CHAR,
                               Account# CHAR,
                               Source CHAR }
                           KEY { Trans#, Account# } ;

CONSTRAINT FK_for_payment_in
           IS_EMPTY ( payment_in NOT MATCHING not_by_cheque ) ;
```

**BR12:**

```
VAR by_DD BASE RELATION { Trans# CHAR,
                          Account# CHAR,
                          Payee CHAR }
                    KEY { Trans#, Account# } ;

CONSTRAINT FK_for_by_DD
           IS_EMPTY ( by_DD NOT MATCHING not_by_cheque ) ;
```

**BR7, BR14:**

```
VAR by_card BASE RELATION { Trans# CHAR,
                            Card# CHAR,
                            Payee CHAR }
                      KEY { Trans#, Card# } ;

CONSTRAINT FK_for_by_card
           IS_EMPTY ( by_card NOT MATCHING debit_card ) ;
```

**BR16:**

```
CONSTRAINT relevant_transaction_exists
           IS_EMPTY ( ( by_card JOIN debit_card )
                      NOT MATCHING not_by_cheque ) ;
```

**BR15:**

```
CONSTRAINT card_extant
           IS_EMPTY ( ( by_card JOIN not_by_cheque
                                JOIN debit_card )
                      WHERE Tdate > Expires ) ;
```

**BR16:**

```
CONSTRAINT every_transaction_of_some_type
           IS_EMPTY ( (transact NOT MATCHING not_by_cheque )
                      NOT MATCHING by_cheque ) ;
```

**BR17:**

```
CONSTRAINT transaction_is_of_only_one_type
           WITH ( bc_dc := by_card JOIN debit_card ) :
           IS_EMPTY ( by_cheque MATCHING not_by_cheque ) AND
           IS_EMPTY ( payment_in MATCHING
                    ( by_DD { Trans#, Account# } UNION
                      BC_DC { Trans#, Account# } ) ) AND
           IS_EMPTY ( by_DD MATCHING BC_DC ) ;
```

**BR18:**

```
CONSTRAINT no_transaction_before_account_open
           ALL ( not_by_cheque JOIN account,
                 DateOpened > Tdate ) ;
```

8. Based on your experiences with Exercise 7, suggest enhancements to **Tutorial D** to make it easier to express any constraints you declared that struck you as being of a common enough kind to warrant an additional shorthand.

The constraint `at_most_one_phone_per_type` could be expressed as a key constraint if we allowed keys to be specified on relational expressions in general rather than just on base relvars in particular. Here we would like to specify `KEY { Customer#, PhoneType }` on `cust_phone JOIN phone_type`. The same shorthand could be used for the constraint `trans#_unique_within_account#`, where we would specify `KEY { Trans#, Account# }` on `by_card JOIN debit_card`.

In the Option 1 solution, the constraint `transaction_is_of_only_one_type` could be expressed more succinctly if we could just specify that the projections of each of the transaction type relvars on `{ Trans#, Account# }` must be disjoint (have no tuples in common)—in other words, that the same combination of `Trans#` and `Account#` values cannot appear in more than one of those relvars. Thus, `{ Trans#, Account# }` would become a kind of key whose uniqueness scope covers more than one relvar.

Note that these shorthands, by raising the level of abstraction in each case, would make the constraints easier to understand as well as perhaps easier to write. Furthermore, they give the DBMS the opportunity to enforce those constraints much more efficiently than is very likely the case if the longhands shown in the given solutions are used.

9. (For students familiar with SQL). Consider the following SQL definitions:

```
CREATE TABLE SF ( StudentId CHAR(4),
                  Faculty VARCHAR(50),
        PRIMARY KEY ( StudentId )
        UNIQUE ( StudentId, Faculty ) ;

CREATE TABLE CF ( CourseId CHAR(4),
                  Faculty VARCHAR(50),
        PRIMARY KEY ( CourseId )
        UNIQUE ( CourseId, Faculty );

CREATE TABLE SCF ( StudentId CHAR(4),
                   CourseId CHAR(4),
                   Faculty VARCHAR(50),
        PRIMARY KEY ( StudentId, CourseId ),
        FOREIGN KEY ( StudentId, Faculty )
                    REFERENCES SF ( StudentId, Faculty ),
        FOREIGN KEY ( CourseId, Faculty )
                    REFERENCES CF ( CourseId, Faculty ) ;
```

a. What problem was the designer solving here?

A constraint was needed to ensure that a combination of StudentId and Faculty values appearing in SCF also appears in SF. A similar constraint was needed to ensure that a combination of CourseId and Faculty values appearing in SCF also appears in CF. The real world situation might, for example, be that each student at the university belongs to exactly on of its faculties, each course is offered by exactly one of its faculties, and a student enrolled on a course must belong to the faculty offering that course.

b. What possible problem remains in this solution?

SCF is not in 5NF and therefore is subject to redundancy. It is not in 5NF because it is not in BCNF, and it is not in BCNF because of the FDs { StudentId } → { Faculty } and { CourseId } → { Faculty }, each of whose determinants is a proper subset of the primary key of SCF.

To normalize SCF all we need to do is drop the Faculty column from the table definition and from each of the foreign keys. But then we would need a constraint to ensure that the Faculty values in the SF and CF rows referenced by a row in SCF are equal. Again, such constraints can be expressed in standard SQL but most SQL implementations do not support any of the standard's features that would make this possible.

c. Describe and comment on the particular features of SQL that make this solution possible.

In standard SQL the required constraints could be expressed like this:

```
CREATE ASSERTION right_faculty_for_course
     CHECK NOT EXISTS ( SELECT StudentId, Faculty
                        FROM SCF
                        EXCEPT
                        SELECT StudentId, Faculty
                        FROM SCF )
```

and similarly for the other foreign key. However, most SQL implementations do not support `CREATE ASSERTION` and in any case such a constraint is unlikely to be enforced efficiently, as foreign key constraints are.

So, in practice the required constraints can only be expressed using the `FOREIGN KEY` construct, probably taking advantage of special indexes created on the relevant columns. But the `FOREIGN KEY` construct requires the referenced columns to constitute a key of the referenced table, specified using either `PRIMARY KEY` or `UNIQUE`. Here our referenced columns constitute a proper superkey in each case. Although the same indexes could be used for efficiency purposes, standard SQL does not allow references to proper superkeys, and nor do any SQL implementations we are aware of.

The well-known "hack" to get around this problem is illustrated in the example. We take advantage of another quirk in SQL, whereby one is permitted to specify a redundant `UNIQUE` constraint. The columns of that redundant `UNIQUE` constraint can then be used in a `FOREIGN KEY` declaration! One might ask, why does the system need to be told that (student id, faculty ) combinations are unique, when it already knows that student ids are unique by themselves?

## 2.8 Additional Exercises Using Rel

1. Create a virtual relvar named `myvars` giving the `Name, Owner,` and `isVirtual` of every relvar not owned by 'Rel'.

```
VAR myvars VIRTUAL ( sys.Catalog WHERE Owner <> 'Rel' )
                   { Name, Owner, isVirtual } ;
```

2. If you haven't already done so, load the file `OperatorsChar.d`, provided in the Scripts subdirectory of the *Rel* program directory, and execute it. One of the relvars mentioned in `sys.Catalog` is named `sys.Operators`. Display the contents of that relvar. How many attributes does it have? What is the declared type of the attribute named `Implementations`?

   Two attributes: `Name` and `Implementations`

   The declared type of `Implementations` is:
   `RELATION {Signature CHARACTER, ReturnsType CHARACTER, Definition CHARACTER, Language CHARACTER, CreatedByType CHARACTER, Owner CHARACTER, CreationSequence INTEGER}`

   Relation types aren't normally recommended for attributes of database relvars, but all updates to the system catalog are performed "under the covers" by the system itself, which should be capable of handling all the difficulties caused by relation-valued attributes. That said, queries against such relvars can be difficult to express, unless you begin them by ungrouping, as suggested in the next exercise.

3. Evaluate the expression

```
(sys.Operators ungroup (Implementations)
where Language = 'JavaF')
{ ALL BUT Language, CreatedByType, Owner, CreationSequence}
```

   What are the "ReturnsTypes" of `LENGTH`, `IS_DIGITS`, and `SUBSTRING`?

   `INTEGER`, `BOOLEAN`, and `CHARACTER`, respectively.

4. Note that if *s* is a value of type CHAR, then LENGTH(*s*) gives the number of characters in *s*, IS_DIGITS(*s*) gives TRUE if and only if every character of *s* is a decimal digit. SUBSTRING(*s*,0,*l*) gives the string consisting of the first *l* characters of *s* (note that strings are considered to start at position 0, not 1). SUBSTRING(*s*,*f*) gives the string consisting of all the characters of *s* from position *f* to the end.

What is the result of IS_DIGITS('')? Is it what you expected? Is it consistent with the definition given above?

TRUE. This is to be expected on the understanding that "everything is true of all elements of the empty set". The string '' contains no characters and therefore does not contain a character that isn't a digit. $(\forall x)P(x)$ is logically equivalent to $\neg(\exists x)\neg P(x)$.

5. Using operators defined by OperatorsChar.d, define types for supplier numbers and part numbers, following Example 2.4 from Chapter 2.

```
TYPE SNO POSSREP { c CHAR CONSTRAINT
                           SUBSTRING(c,0,1) = 'S' AND
                           IS_DIGITS(SUBSTRING(c,1)) } ;
TYPE PNO POSSREP { c CHAR CONSTRAINT
                           SUBSTRING(c,0,1) = 'P' AND
                           IS_DIGITS(SUBSTRING(c,1)) } ;
```

Define relvars Srev, Prev, and SPrev as replacements for S, P and SP, using the types you have just defined as the declared types of attributes S# and P#.

```
VAR Srev BASE RELATION {S# SNO, Sname CHAR,
                        Status INTEGER, City CHAR}
                   KEY { S# } ;
VAR Prev BASE RELATION {P# PNO, Pname CHAR, Colour CHAR,
                        Weight RATIONAL, City CHAR}
                   KEY { P# } ;
VAR SPrev BASE RELATION {S# SNO, P# PNO, Qty INTEGER}
                    KEY { S#, P# } ;
```

Write relvar assignments to copy the contents of S, P and SP to Srev, Prev, and SPrev, respectively. Note that if SNO is the type name for supplier numbers in S and Srev, then SNO(S#) "converts" an S# value in S to one for use in Srev.

We need to use EXTEND to add an attribute to contain the "converted" S# and/or P# values. Happily, the exercise is much easier now, with **Tutorial D** Version 2, because we can use the existing attribute names S# and P# in the "extensions", which actually become replacements:

```
Srev := EXTEND S : { S# := SNO(S#) } ;

Prev := EXTEND P : { P# := PNO(P#) } ;

SPrev := EXTEND SP : { S# := SNO(S#), P# := PNO(P#) } ;
```

In each case the S# or P# attribute is replaced by one of type SNO or PNO, respectively, with values in each tuple obtained by evaluation of the specified invocation of the SNO or PNO selector. Thus, we are replacing an existing relvar attribute, rather than adding one.

6. Using the relvars defined in Exercise 5, repeat Exercise 6 from the set headed "Working with A Database in *Rel*" given with the exercises for Chapter 4. Which of your solutions need revisions beyond the obvious changes in relvar names?

b. Get supplier numbers for suppliers who supply part P1.

```
S MATCHING ( SP WHERE P# = PNO('P1') ) { S# }
```

h. Get supplier numbers for suppliers with a status lower than that of supplier S1.

```
( S JOIN ( ( ( S WHERE S# = SNO('S1') ) { Status }
                  RENAME { Status AS S1_Status }
                )
              )
     WHERE Status < S1_Status
   ) { S# }
```

Note that my solution to Exercise f. in this set uses "<" to compare supplier numbers. The fact that this solution still works when supplier numbers are of type SNO instead of CHAR tells us that *Rel* treats SNO as an ordered type, the ordering being based on that of the declared type, CHAR, of its only possrep component.