CHAPTER 4

Advanced Aspects

CONTENTS

If you have completed the first three chapters of this book, you should feel comfortable programming in LogiQL. In this chapter, you will learn about some other aspects of the language useful for specialized purposes. The first unit illustrates the power and scope of the language by describing how to use LogiQL to implement some features of imperative programming languages. Some additional constraints are presented in the second unit. The third unit introduces an advanced technique, called *derived entities*, useful in situations where your knowledge of a particular entity comes from its properties. Although most of the examples you have seen in this book are relatively small, LogiQL can be used to develop industrial-scale applications. With this in mind, Unit 4 describes some techniques you can use for structuring large programs. The last unit presents three topics (derived-only views, hierarchical syntax, and file predicates) that do not warrant a unit of their own, but that can nevertheless be quite useful in certain circumstances. Finally, the chapter ends with a consolidation exercise giving you an opportunity to practice the skills you have learned in the chapter.

UNIT 4.1: EMULATING IMPERATIVE PROGRAMMING CONSTRUCTS

Imperative programming languages like Java include various control structures for carrying out tasks in a procedural rather than declarative fashion. These include alternation or selection constructs for choosing one action from various alternatives (e.g., **if-then, if-then-else,** and **case/switch** statements) and iteration constructs for looping through some code either a specified number of times or while/until some condition applies (e.g., **for** and **while** statements). In this unit we discuss how to emulate these structures in LogiQL.

To help explain some of the techniques, we use *flowcharts* to provide simple diagrams of their control flows. The non-programming example shown in Figure 4.1 is a flowchart for the following procedure: *if you are tired, then go to sleep*. Here a diamond shape depicts a decision box for some condition, and a rectangular shape depicts a command box for some instructions. The condition indicated in the decision box corresponds to a proposition (e.g., you are tired), so it is either true or false but not both. Flowlines with arrowheads direct the control flow from one step to the next, and the flow-annotations "T" and "F" indicate which way to move when the condition is true or false, respectively. The command box includes one or more sentences, each of which expresses a command, instruction, or action (e.g., go to sleep) to be carried out.

Figure 4.2 generalizes this example, using *c* to denote the condition and *s* to denote the programming statement to be carried out. In an imperative language like Java, this is called an **if** statement. In imperative programming, the term *statement* is used to indicate an instruction rather than a proposition, so is somewhat misleading, since in ordinary usage statements express propositions.

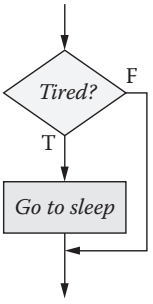


FIGURE 4.1 Nighttime flowchart.

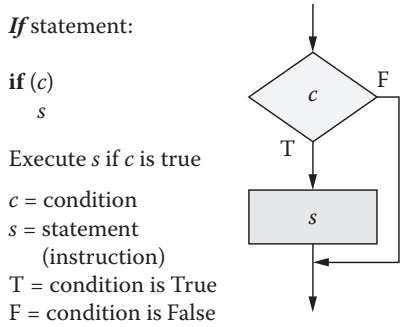


FIGURE 4.2 Flowchart for *if* statement.

LogiQL is a declarative language, so its statements or formulas are used to declare propositions rather than instructions. For example, the following derivation rule declares that *if p1 is male and is a sibling of p2 then p1 is a brother of p2*:

```
isBrotherOf(p1, p2) <- isMale(p1), isSiblingOf(p1, p2).
```

The left arrow "<-" denotes the propositional operator **if**. You may take this to be the case even for delta rules. For example, the following delta rule may be read propositionally as follows: *if it is added that p1 is male and it is added that p1 is a sibling of p2, then it is added that p1 is a brother of p2*. If you were instead to read the "+" in the rule head as "add" rather than "it is added that," then the rule head would effectively correspond to a command rather than a proposition:

```
+isBrotherOf(p1, p2) <- +isMale(p1),
    +isSiblingOf(p1, p2).
```

If Statements

In imperative programming languages, the **if-then-else** statement has the semantics shown in the Figure 4.3 flowchart. If the condition *c* is true, execute statement s_1; otherwise, execute statement s_2. As a non-programming example, the following instruction fits this pattern: *if you are tired then go to bed, else go for a walk*.

In everyday life, we often encounter statements of the form *if p then q else r*, where *p*, *q*, and *r* all denote propositions. For example: *if you score at least half the total points on the exam, then you pass, else you fail.*

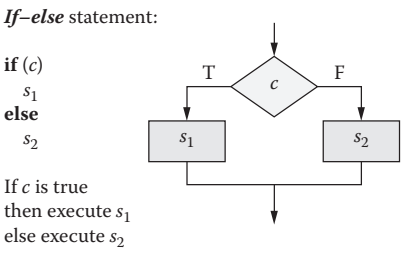


FIGURE 4.3 Flowchart for *if-else* statement.

TABLE 4.1 Student Names

Student Number	Given Name	Family Name	Full Name
101	John	Smith	John Smith
102	Ann	Jones	Ann Jones
103		Ah	Ah
104	John	Smith	John Smith
…	…	…	…

LogiQL does not support this construct directly, so we instead emulate it using a pair of rules as shown below:

Construct	LogiQL
if *p* **then** *q*	`q <- p.  // q if p.`
else *r*	`r <- !p.  // r if not p.`

As a simple example, consider Table 4.1 about students at a university. Students are identified by their student number. All students have a family name. Although most students have a given name, it is possible for some students to have no given name. (Although rare, this can be the case, especially for international students.)

Our task is to write a rule to derive the full names of students from their given name (if any) and family name. In cases like this, it often helps to first write down the syntax of the required term. The notation we use is Extended Backus-Naur Form (EBNF), which is described in Appendix C. For now, we use "`::=`" to denote "is defined as" and append a question mark "`?`" to an item that is optional. Here, the parentheses delimit a list composed of a given name followed by a space character:

```
FullName ::= (GivenName " ")? FamilyName.
```

So a full name optionally starts with a given name followed by a space character, and must end with a family name. Using "+" as the string concatenation operator, it is natural to phrase this rule informally in terms of an **if-the-else** statement; thus,

```
if the student has a given name
 then the fullname is the given name + " " + the
   family name
 else the fullname is the family name.
```

Replacing the **if-then-else** pattern by two **if** rules as discussed above, we may now code the task in LogiQL:

```
// Schema
Student(s), hasStudentNr(s:n) -> int(n).
givenNameOf[s] = gn -> Student(s), string(gn).
familyNameOf[s] = fn -> Student(s), string(fn).
fullNameOf[s] = gn -> Student(s), string(gn).
fullNameOf[s] = gn + " " + familyNameOf[s] <-
givenNameOf[s] = gn.
fullNameOf[s] = familyNameOf[s] <- !givenNameOf[s] = _.
// If student s has the givenName gn
// then the fullName of s is gn + " " + the
// familyName of s
// else the fullName of s is the familyName of s.

// Data
+givenNameOf[101] = "John",
    +familyNameOf[101] = "Smith".
+givenNameOf[102] = "Ann", +familyNameOf[102] = "Jones".
+familyNameOf[103] = "Ah".
+givenNameOf[104] = "John",
    +familyNameOf[104] = "Smith".
```

Querying the predicate `fullNameOf` yields the following result. The program and data are available in the files `FullName1.logic` and `FullName1Data.logic`, respectively:

```
104, John Smith
103, Ah
102, Ann Jones
101, John Smith
```

Now let's extend the example to also record a student's second given name, if any. Table 4.2 includes a sample population.

TABLE 4.2 Student Names and Numbers

Student Number	Given Name 1	Given Name 2	Family Name	Full Name
101	John	Thomas	Smith	John Thomas Smith
102	Ann	Linda	Jones	Ann Linda Jones
103			Ah	Ah
104	John		Smith	John Smith
…	…		…	…

The syntax for full names of students may now be set out as follows:

```
FullName ::= (GivenName1 " " (GivenName2 " ")?)?
    FamilyName.
```

The derivation rule for full names could now be stated informally as the following nested **if-then-else** statement:

```
if the student has givenname1
  then if the student has givenname
    then fullname is givenname1 + " " + givenname2 +
      " " + family name
    then fullname is givenname1 + " " + family name
else fullname is the family name.
```

Note that the family name is appended in each of the three alternatives. To avoid such repetition, it is better to split the task into two parts: Derive the list of given names (if any), and then append the full name. One way to code this is as follows:

```
// Schema
Student(s), hasStudentNr(s:n) -> int(n).
givenName1Of[s] = gn -> Student(s), string(gn).
givenName2Of[s] = gn -> Student(s), string(gn).
familyNameOf[s] = fn -> Student(s), string(fn).
givenName2Of[s] = _ -> givenName1Of[s] = _.
// If s has a 2nd given name then s has a first
// givenName.
givenNamesOf[s] = gn1 -> Student(s), string(gn1).
givenNamesOf[s] = gn1 + " " + gn2 + " " <-
    givenName1Of[s] = gn1, givenName2Of[s] = gn2.
givenNamesOf[s] = gn1 + " " <-
```

```
    givenName1Of[s] = gn1, !givenName2Of[s] = _.
fullNameOf[s] = gns -> Student(s), string(gns).
fullNameOf[s] = gns + familyNameOf[s] <-
    givenNamesOf[s] = gns.
fullNameOf[s] = familyNameOf[s] <- !givenNamesOf[s] = _.

// Data
+givenName1Of[101] = "John",
    +givenName2Of[101] = "Thomas",
+familyNameOf[101] = "Smith".
+givenName1Of[102] = "Ann",
    +givenName2Of[102] = "Linda",
+familyNameOf[102] = "Jones".
+familyNameOf[103] = "Ah".
+givenName1Of[104] = "John",
    +familyNameOf[104] = "Smith".
```

Querying the predicate `fullNameOf` yields the following result. The program and data are available in the files `FullName2.logic` and `FullName2Data.logic`:

```
104, John Smith
103, Ah
102, Ann Linda Jones
101, John Thomas Smith
```

Switch Statement

As a generalization of the **if-then-else** statement, most imperative programming languages provide a **switch** statement (or **case** statement) for choosing one of many options based on the value of some expression. A flowchart for this statement is shown in Figure 4.4, along with the basic statement syntax used in Java (typically a **break** statement is included after each case option to exit immediately if that option is executed). In the case where the expression e evaluates to the value v_1, statement s_1 is executed. If e evaluates to v_2, then s_2 is executed, and so on. If e evaluates to none of the values $v_1 \ldots v_n$, then the default statement s_{n+1} is executed.

As a simple example, consider the report shown in Table 4.3 about prices of expensive items for a store chain. Customers may pay an annual membership fee to enroll as members of a discount program at one of three levels (bronze, silver, gold). The base price for an item is the price to non-members.

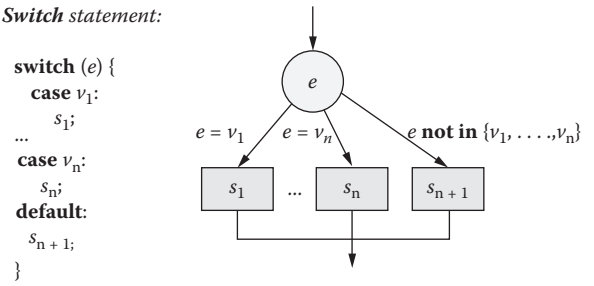


FIGURE 4.4 Flowchart for the *switch* statement.

TABLE 4.3 Discount Program Data

Item Code	Base Price (US$)	Bronze-Level Price (US$)	Silver-Level Price (US$)	Gold-Level Price (US$)
A1	100.00	95.00	90.00	80.00
B2	200.00	195.00	180.00	160.00
C3	100.00	95.00	90.00	80.00
D4	100.90	95.90	90.90	80.90

The rule for determining prices for members at various levels may be specified in pseudocode as follows, using a simpler cases syntax, "`:=`" for assignment, and assuming prices are in U.S. dollars (US$). The `floor` function truncates a number to remove any digits after the decimal point, for example, `floor(80.72) = 80`:

```
cases for memberLevel
 bronze: price := basePrice - 5.0
 silver: price := floor(basePrice * 0.9)
 gold: price := floor(basePrice * 0.8)
else price := basePrice
```

Algorithms like this may be easily emulated in LogiQL using multiple rules, one for each case option. The following program and data show one way to compute the report. The function `floor[n]` is a built-in function in LogiQL. Note that the three rules have a head but no body and that argument of the `floor` function is itself a function expression:

```
// Schema
Item(i), hasItemCode(i:c) -> string(c).
Item(i) -> basePriceOf[i] = _.
Level(l), hasLevelName(l:n) -> string(n).
```

```
priceOf_AtLevel_[i, l] = n -> Item(i), Level(l),
    float(n).
basePriceOf[i] = p -> Item(i), float(p).
priceOf_AtLevel_[i, "Bronze"] = basePriceOf[i] - 5.0f.
priceOf_AtLevel_[i, "Silver"] = floor[basePriceOf[i]
    * 0.9f].
priceOf_AtLevel_[i, "Gold"] = floor[basePriceOf[i]
    * 0.8f].

// Data
+Level("Bronze").
+Level("Silver").
+Level("Gold").

+basePriceOf["A1"] = 100.0f.
+basePriceOf["B2"] = 200.0f.
+basePriceOf["C3"] = 100.0f.
+basePriceOf["D4"] = 100.90f.
```

The following query returns the result shown, to reproduce the data in the report:

```
_(i, bp, brp, sp, gp) <-
    basePriceOf[i] = bp,
    priceOf_AtLevel_[i, "Bronze"] = brp,
    priceOf_AtLevel_[i, "Silver"] = sp,
    priceOf_AtLevel_[i, "Gold"] = gp.
```

Result:

```
D4, 100.9, 95.9, 90, 80
C3, 100, 95, 90, 80
B2, 200, 195, 180, 160
A1, 100, 95, 90, 80
```

This solution may be easily extended to include members and their membership levels and to determine item prices for them based on their membership level. The program and data are available in the files `Cases.logic` and `CasesData.logic`.

Iteration Statements

While the above approach can be extended to handle students with more than two given names, it would require a lot of code to deal with lengthy

TABLE 4.4 Given Names and Houses of British Monarchs

Monarch	Given Names	House
Anne	Anne	Stuart
George I	George, Louis	Hanover
George II	George, Augustus	Hanover
George III	George, William, Frederick	Hanover
George IV	George, Augustus, Frederick	Hanover
William IV	William, Henry	Hanover
Victoria	Alexandrina, Victoria	Hanover
Edward VII	Albert, Edward	Saxe-Coburg and Gotha
George V	George, Frederick, Ernest, Albert	Windsor
Edward VIII	Edward, Albert, Christian, George, Andrew, Patrick, David	Windsor
George VI	Albert, Frederick, Arthur, George	Windsor
Elizabeth II	Elizabeth, Alexandra, Mary	Windsor

lists of given names. For example, recall the facts shown in Table 4.4 about British monarchs, in which Edward VIII has seven given names.

Suppose we need to derive the full name of a monarch by appending the house name of the monarch to his/her list of given names. The syntax for full names of monarchs may be set out as follows. Here, appending an asterisk ("`*`") to an item indicates zero or more occurrences of that item. Since each monarch has at least one given name, we do not need to cater to the case of no given name:

```
GivenNames ::= GivenName (" " GivenName)*.
FullName ::= GivenNames " " HouseName.
```

We could set an upper limit (e.g., 10) on the number of given names for a monarch and use different predicates (`givenName1Of`, `givenName2Of`, …, `givenName10Of`) for each given name position. However, it is simpler to use a single ternary predicate that includes the position of the given name in the list:

```
givenNameOf_AtPosition_[m, n] = gn -> Monarch(m),
    int(n), string(gn).
```

This functional naming style uses underscores ("`_`") in the predicate name to indicate the placeholders for the function's arguments (in this case `m` and `n`). So you can read the function declaration as *given name of m at position n = gn*. Apart from being simpler, using this ternary predicate

is more flexible than using multiple binary predicates, because it caters for cases where it is impractical to set an upper limit on the size of the given-name list. The derivation rule for given names could now be stated informally as the following imperative pseudocode, using ":=" for *becomes* or "*is assigned the value of*":

```
for each monarch m
    n : = count of given names of m
    givenNames of m : = givenName1 of m
  for i : = 2 to n do
      givenNames of m : = givenNames of m + " " +
          givenName[i] of m
```

Here, for each monarch we use the `count` function to determine the number of given names of the monarch. Then we initialize `givenNames` to the first given name. Finally, we use a **for** loop to iterate over the rest of the names, appending them one at a time until we have appended the last given name.

A flowchart for a general **for** loop is shown in Figure 4.5, using ":=" for the assignment operator. As usual, *s* indicates a programming statement. The statement syntax at the top is from the Pascal language, which was also used in our pseudocode. The lower statement syntax is that used in Java, where "=" is used for assignment, and "++" is used to increment the value of a variable by 1.

Although deriving lists of given names for monarchs is naturally conceived imperatively in terms of the algorithm specified above in pseudocode, LogiQL does not provide iterative control structures like **for** loops.

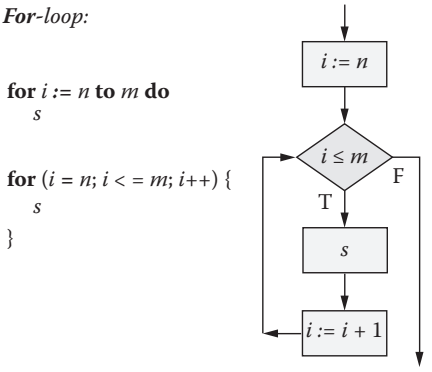


FIGURE 4.5 Flowchart for a *for* loop.

Happily, such iterative procedures can be reformulated declaratively in LogiQL by using recursion.

For any given monarch `m`, we compute `nrOfGivenNamesOf[m]`, the number of given names of `m`, using the `count` aggregation function as shown below. We then recursively derive `givenNamesOf_ToPosition_[m,n]`, the list of `m`'s first `n` given names, in multiple steps. First, we use `m`'s first given name as the basis for position `n = 1`, then we append a space and the given name at position `n` (where `n > 1`) to `givenNamesOf_ToPosition_[m,n-1]`, the list of `m`'s first `n–1` given names. Doing this recursively calls the function until the value of `n–1` decrements to the basis value `1`. The functional predicate `givenNamesOf[m]` may now be computed by setting `n` in the functional predicate `givenNamesOf_ToPosition_[m,n]` to equal the number of given names of `m`. Finally, the full name of `m` is derived by appending a space and the house name of `m` to `m`'s given name list:

```
Monarch(m), hasMonarchName(m:mn) -> string(mn).
House(h), hasHouseName(h:hn) -> string(hn).
houseOf[m] = h -> Monarch(m), House(h).
givenNamesOf_ToPosition_[m, n] = gn -> Monarch(m),
    int(n), string(gn).
givenNameOf_AtPosition_[m, n] = gn -> Monarch(m),
    int(n), string(gn).
Monarch(m) -> givenNameOf_AtPosition_[m, _] = _.
// Each Monarch has a given name.
nrOfGivenNamesOf[m] = n -> Monarch(m), int(n).
nrOfGivenNamesOf[m] = n <-
    agg<<n = count()>>
    givenNameOf_AtPosition_[m, _] = _.

givenNamesOf_ToPosition_[m, 1] = gn <-
    givenNameOf_AtPosition_[m, 1] = gn.
// Basis clause: first given name of monarch m.

givenNamesOf_ToPosition_[m, n] = gns + " " + gn <-
    givenNameOf_AtPosition_[m, n] = gn, n > 1,
    givenNamesOf_ToPosition_[m, n-1] = gns.
// Recursive clause: first n names = first n-1 names
// + nth name.
```

```
givenNamesOf[m] = gns -> Monarch(m), string(gns).
givenNamesOf[m] = gns <-
    nrOfGivenNamesOf[m] = n,
    givenNamesOf_ToPosition_[m, n] = gns.

fullNameOf[m] = gns -> Monarch(m), string(gns).
fullNameOf[m] = gns + " " + hn <-
    givenNamesOf[m] = gns,
    houseOf[m] = h, hasHouseName(h:hn).
```

The first few lines of the data file are shown below. To save space, data for only the first two monarchs are shown here:

```
// Data
+givenNameOf_AtPosition_["Anne", 1] = "Anne",
+houseOf["Anne"] = "Stuart".
+givenNameOf_AtPosition_["George I", 1] = "George",
+givenNameOf_AtPosition_["George I", 2] = "Louis",
+houseOf["George I"] = "Hanover".
etc.
```

Querying the predicate `fullNameOf` yields the following result. Note that with the current page width and font size, the full name for Edward VIII spills over onto a second line. The program and data are available in the files `FullName3.logic` and `FullName3Data.logic`:

```
Elizabeth II, Elizabeth Alexandra Mary Windsor
George VI, Albert Frederick Arthur George Windsor
Edward VIII, Edward Albert Christian George Andrew
    Patrick
David Windsor
George V, George Frederick Ernest Albert Windsor
Edward VII, Albert Edward Saxe-Coburg and Gotha
Victoria, Alexandrina Victoria Hanover
William IV, William Henry Hanover
George IV, George Augustus Frederick Hanover
George III, George William Frederick Hanover
George II, George Augustus Hanover
George I, George Louis Hanover
Anne, Anne Stuart
```

Above we used a **for** loop for the imperative procedure, because the number of iterations to be performed was known from counting. Alternatively, the **for** loop could have been replaced by the following pseudocode, which instead makes use of a **while** loop:

```
i : = 2
while i < = n do
givenNames of m : = givenNames of m + " " +
    givenName[i] of m
i : = i + 1
```

A flowchart for a **while** loop is shown in Figure 4.6. If the condition *c* is false, the loop instruction *s* is never executed. A **do-until** loop (called **repeat-until** in Pascal) and a **do-until-not** loop (misleadingly called a **do-while** loop in Java) place the condition last, so always execute the loop instruction at least once. Depending on the condition, such loops can be emulated using recursion in a similar way to that discussed above.

Tip: Emulate **if-then-else** and **switch/case** constructs by using multiple rules, and emulate iterative loops via recursion.

Exercise 1A: The report in the Table 4.5 extract identifies famous authors by their author number, and also records the given names and family names that they used as authors. Each author has between one and three given names.

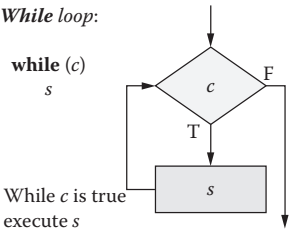


FIGURE 4.6 Flowchart for a *while* loop.

TABLE 4.5 Author Names

Author Number	Given Name 1	Given Name 2	Given Name 3	Family Name
1	John	Ronald	Reuel	Tolkien
2	Isaac			Asimov
3	Joanne	Kathleen		Rowling
…	…		…	…

The stub of a LogiQL program and data for this report are provided below and are accessible in the file `AuthorNames.logic` and `AuthorNamesData.logic`. Extend the program to derive the author full names without using recursion:

```
// Schema
Author(a), hasAuthorNr(a:n) -> int(n).
givenName1Of[a] = gn -> Author(a), string(gn).
givenName2Of[a] = gn -> Author(a), string(gn).
givenName3Of[a] = gn -> Author(a), string(gn).
familyNameOf[a] = fn -> Author(a), string(fn).
Author(a) -> givenName1Of[a] = _, familyNameOf[a] = _.
givenName2Of[a] = _ -> givenName1Of[a] = _.
givenName3Of[a] = _ -> givenName2Of[a] = _.

// Data
+givenName1Of[1] = "John", +givenName2Of[1] = "Ronald",
    +givenName3Of[1] = "Reuel",
    +familyNameOf[1] = "Tolkien".
+givenName1Of[2] = "Isaac", +familyNameOf[2] = "Asimov".
+givenName1Of[3] = "Joanne",
    +givenName2Of[3] = "Kathleen",
    +familyNameOf[3] = "Rowling".
```

Exercise 1B: In mathematics, *factorial n*, written as *n*!, where *n* is a whole number, is defined as follows: if $n = 0$, $n! = 1$; if $n > 0$ then $n! = 1 \times \ldots \times n$ (i.e., the product of all natural numbers up to *n*). For example, Table 4.6 lists the factorials of the first six whole numbers.

TABLE 4.6 Factorial Function

n	*n!*
0	1
1	1 (i.e., 1×1)
2	2 (i.e., 1×2)
3	6 (i.e., $1 \times 2 \times 3$)
4	24 (i.e., $1 \times 2 \times 3 \times 4$)
5	120 (i.e., $1 \times 2 \times 3 \times 4 \times 5$)
..	..

An iterative procedure that uses a **for loop** to derive x = factorial n for any whole number n may be specified as the following pseudocode, using ":=" for assignment:

```
x : = 1
for i : = 2 to n do
  x : = x * i
return x
```

Write a LogiQL program that uses recursion to derive the function `factorial[n]=x`. Test your program for values of `n` in the range 0 .. 20. A program stub that includes the following code is accessible as `Factorial.logic`. Complete the program by adding your rules to compute `factorial[n]`. *Hint:* To ensure that your recursive clause is safe, you need to restrict the values of `n` to just the values in the range. You can use the `int:range` function to do this. Its arguments are, respectively, the first element in the range, the last element, the increment between elements, and the variable into which the generated values should be placed:

```
factorial[n] = x -> int(n), int(x).
SmallInteger(n) -> int(n).
SmallInteger(n) <- int:range(0, 20, 1, n).
```

Exercise 1C: A test with possible scores of 0 through 10 is given to students. An extract of a report containing the test results is shown in Table 4.7. Students are identified by their student numbers.

Letter grades for the test are determined by the following procedure specified in pseudocode:

```
cases for testScore
  10: grade : = 'A'
  7, 8, 9: grade : = 'B'
```

TABLE 4.7 Student Grades

Student Number	Score	Letter Grade
101	7	B
102	9	B
103	5	C
104	10	A
105	4	F
106	7	B

```
  5, 6: grade : = 'C'
  else grade : = 'F'
```

The stub of a program and data for this report, available as `Grades.logic` and `GradesData.logic`, include the following code:

```
// Schema
Student(s), hasStudentNr(s:n) -> int(n).
scoreOf[s] = n -> Student(s), int(n).
scoreOf[_] = n -> n > = 0, n < = 10.

// Data
+scoreOf[101] = 7, +scoreOf[102] = 9, +scoreOf[103] = 5.
+scoreOf[104] = 10, +scoreOf[105] = 4, +scoreOf[106] = 7.
```

Extend the program to assign letter grades to the students.

UNIT 4.2: FURTHER CONSTRAINTS

To help ensure that data entered in the workspace agree with the application domain being modeled, it is important to include code to enforce various constraints that apply in that domain. Previously we discussed how to code various kinds of constraints in LogiQL, such as uniqueness constraints, mandatory role constraints, subset constraints, simple exclusion constraints, and ring constraints. This unit discusses how to encode several other kinds of constraints, including equality constraints, further value constraints, frequency constraints, and subset and exclusion constraints

Equality Constraints

Consider Table 4.8 containing blood pressure (BP) readings for hospital patients. Patients are identified by patient numbers. Blood pressure readings involve two measurements: the systolic BP is the maximum pressure during a heartbeat, and the diastolic BP is the minimum pressure.

TABLE 4.8 Patient Blood Pressure (BP) Readings

Patient Number	Systolic BP (mm Hg)	Diastolic BP (mm Hg)
1001	120	80
1002	135	95
1003	120	80
1004		

A basic schema to capture the data is shown below. For simplicity, the code assumes that the pressure unit (millimeters of mercury) is understood:

```
Patient(p), hasPatientNr(p:n) -> int(n).
systolicBPof[p] = n -> Patient(p), int(n).
diastolicBPof[p] = n -> Patient(p), int(n).
```

If instead you wish to explicitly declare pressure and its unit, replace the last two lines of code above by the following:

```
Pressure(pr), has_mmHgValue(pr:n) -> int(n).
systolicBPof[p] = pr -> Patient(p), Pressure(pr).
diastolicBPof[p] = pr -> Patient(p), Pressure(pr).
```

Either way, the data for the table may be entered as follows:

```
+systolicBPof[1001] = 120, +diastolicBPof[1001] = 80.
+systolicBPof[1002] = 135, +diastolicBPof[1002] = 95.
+systolicBPof[1003] = 120, +diastolicBPof[1003] = 80.
+Patient(1004).
```

Not all patients might have their BP taken (e.g., patient 1004 has not been tested yet), but if the BP is taken, both the diastolic and systolic readings are needed. Hence, the population of patients who have their systolic BP taken must *equal* the population of patients who have their diastolic BP taken. This is a simple example of an **equality constraint**.

An equality constraint between two roles is equivalent to two subset constraints between the roles, one constraint in each direction. So the equality constraint above may be coded as follows:

```
systolicBPof[p] = _ -> diastolicBPof[p] = _.
diastolicBPof[p] = _ -> systolicBPof[p] = _.
// If patient p has systolic BP then p has diastolic
// BP.
// If patient p has diastolic BP then p has systolic
// BP.
```

To check that the constraint is enforced properly, you can try either of the following updates. Each should generate a constraint violation error message, because if the update were accepted, the relevant patient would have only one of the two pressure readings:

```
+diastolicBPof[1004] = 90. // Error (no systolic BP)!
-diastolicBPof[1001] = 80. // Error (no diastolic BP)!
```

As an example of an equality constraint between pairs of roles, suppose that we wish to record the history of patient blood pressures, where a patient may have at most one BP reading per day. The functional, ternary predicates to record the relevant pressure `n` for patient `p` on date `d` may be declared as follows:

```
systolicBPfor[p, d] = n -> Patient(p), datetime(d),
    int(n).
diastolicBPfor[p, d] = n -> Patient(p), datetime(d),
    int(n).
```

The equality constraint between the populations of (`p, d`) pairs may now be coded as follows:

```
systolicBPfor[p, d] = _ -> diastolicBPfor[p, d] = _.
diastolicBPfor[p, d] = _ -> systolicBPfor[p, d] = _.
// If patient p has a systolic BP on date d
// then p has a diastolic BP on date d.
// If patient p has a diastolic BP on date d
// then p has a systolic BP on date d.
```

An equality constraint between two or more compatible roles requires the populations of those roles to be equal. More generally, equality constraints can apply between sequences of compatible roles. In either case, an equality constraint may always be enforced by multiple subset constraints.

Value Constraints

Recall that a value constraint restricts the values that may be assigned to an argument of a predicate. Previously, we met some simple value constraints that restrict an argument to an enumerated list of values. For example, we may constrain color codes to be `"R"`, `"G"`, or `"B"`, and thus,

```
Color(c), hasColorCode(c:cc) -> string(cc).
hasColorCode(_:cc) -> cc = "R" ; cc = "G" ; cc = "B".
```

In addition to an enumerated list of values, a value constraint may constrain an argument's value to lie within one or more value ranges. A range with a single bound may be expressed as a simple comparison. For example, if we assume that ages are measured in years, the following constraint confines a person's age to be at most 140 years:

```
ageOf[_] = n -> n <= 140.
```

A range bounded at both ends may be expressed as a conjunction of simple comparisons. For example, the following code constrains product ratings to integers in the range 1..5:

```
productRatingOf[_] = n -> n >= 1, n <= 5.
```

To constrain a value to lie within one of multiple ranges, use a disjunction of expressions, one for each range. For example, the following code constrains the extreme score of an item to be either at least 95 or a non-negative number less than 20. Since conjunction has priority over disjunction, parentheses are not needed around the conjunction that is the second disjunct, although you may include them if you wish:

```
extremeScoreOf[_] = n -> n > 95 ; n >= 0, n < 20.
```

Frequency Constraints

Recall that an internal uniqueness constraint on a single role of a predicate ensures that each instance in the population of that role is unique. Internal uniqueness constraints can also be applied to a list of roles in a predicate. An external uniqueness constraint is a restriction on roles from two or more predicates. In general, a uniqueness constraint on a list of one or more roles ensures that, at any given time, each instance of that role list appears there at most once. A **frequency constraint** generalizes the notion of uniqueness constraint, allowing the number of times the constrained role list appears in any given population to be set to any positive integer, or even to a range of positive integers.

Consider, for example, Tables 4.9 and 4.10 contain reports from a paper-review system. Reviewers are identified by name and papers by number.

TABLE 4.9 Paper Reviewers

Reviewer	Papers Assigned
Ann Jones	1, 2
Bill Smith	1
Cate Wong	1, 2
Dan Green	
Eve Noon	1, 2
Fred Jones	

TABLE 4.10 Unassigned Papers

Unassigned Papers
3

The first report lists the reviewers and the papers they have so far been assigned to review. The latter indicates papers not yet assigned for review. A basic schema for these reports is as follows:

```
Reviewer(r), hasReviewerName(r:rn) -> string(rn).
Paper(p), hasPaperNr(p:n) -> int(n).
isAssigned(r, p) -> Reviewer(r), Paper(p).
```

The schema may be populated with the data shown by executing these updates:

```
+isAssigned("Ann Jones", 1), +isAssigned("Ann Jones", 2).
+isAssigned("Bill Smith", 1).
+isAssigned("Cate Wong", 1), +isAssigned("Cate Wong", 2).
+Reviewer("Dan Green").
+isAssigned("Eve Noon", 1), +isAssigned("Eve Noon", 2).
+Reviewer("Fred Jones").
+Paper(3).
```

Now suppose we wish to ensure that each reviewer is assigned at most two papers to review. The current data satisfy this frequency constraint, but there is nothing to stop us updating the database to violate the constraint (e.g., assert that Ann Jones is assigned paper 3). To code the frequency constraint that each reviewer is assigned at most two papers, we use the `count` function to count the positive number of papers assigned to reviewers, and then constrain that number to be at most 2:

```
positiveNrPapersAssignedTo[r] = n -> Reviewer(r), int(n).
positiveNrPapersAssignedTo[r] = n <-
    agg<<n = count()>> isAssigned(r, _).
// If r is assigned a paper to review, then the number
// of papers assigned to r is the count of its
// assignments.
positiveNrPapersAssignedTo[_] = n -> n <= 2.
// Each reviewer is assigned at most 2 papers to
// review.
```

The use of "`positive`" in the name of the derived function clarifies that we are interested only in reviewers who have been assigned at least one paper to review. Here, the `count` function returns nothing (rather than 0) for reviewers with no paper assignments. For example,

querying the positiveNrPapersAssignedTo predicate returns the following result:

```
Eve Noon, 2
Cate Wong, 2
Bill Smith, 1
Ann Jones, 2
```

Now suppose that we wish to ensure that each paper that is assigned for review is assigned to at least three reviewers. The current data satisfy this constraint but would be violated if we retracted the fact that Ann Jones is assigned paper 2. We may code this constraint using a count function to count the positive number of reviewers for each paper:

```
positiveNrReviewersOf[p] = n -> Paper(p), int(n).
positiveNrReviewersOf[p] = n <-
    agg<<n = count()>> isAssigned(_, p).
// If paper p is assigned a reviewer, then the number
// of reviewers assigned to p is the count of its
// assignments.
positiveNrReviewersOf[_] = n -> n >= 3.
// Each assigned paper is assigned at least 3
// reviewers.
```

Querying the positiveNrReviewersOf predicate returns the following result:

```
1, 4
2, 3
```

The code for the above program (including both frequency constraints) and data, as well as a counterexample test are available in the files PaperReview*.logic.

If the number specified in the frequency constraint is small (e.g., 2 or 3), the frequency constraint may often be coded explicitly, without using the count function. However, if the number is higher, using the count function is more convenient and requires far less code.

The frequency constraints just considered are *internal frequency constraints*, because each applies to a single predicate. An *external frequency constraint* applies to roles from different predicates and is a generalization of an external uniqueness constraint. As an example, consider the following extract from a report shown in Table 4.11 about enrollments of

TABLE 4.11 Student Course Enrollment

Enrollment Number	Student Number	Course Code
1001	120	CS100
1002	120	CS135
1003	501	CS135
3001	120	CS100

students in courses. Enrollments are identified by enrollment numbers, students by student numbers, and courses by codes.

A basic schema for this report may be coded as follows:

```
Enrollment(e), hasEnrollmentNr(e:n) -> int(n).
Student(s), hasStudentNr(s:n) -> int(n).
Course(c), hasCourseCode(c:cc) -> string(cc).
studentInvolvedIn[e] = s -> Enrollment(e), Student(s).
courseInvolvedIn[e] = c -> Enrollment(e), Course(c).
// Each Enrollment involves both a student and a
// course.
Enrollment(e) ->
    studentInvolvedIn[e] = _, courseInvolvedIn[e] = _.
```

The data for this report may be entered as follows:

```
+studentInvolvedIn[1001] = 120,
    +courseInvolvedIn[1001] = "CS100".
+studentInvolvedIn[1002] = 120,
    +courseInvolvedIn[1002] = "CS135".
+studentInvolvedIn[1003] = 501,
    +courseInvolvedIn[1003] = "CS135".
+studentInvolvedIn[3001] = 120,
    +courseInvolvedIn[3001] = "CS100".
```

Now suppose that no student may enroll more than twice in the same course. (This is common practice at many universities, where one is excluded from a course if one fails it twice.) This frequency constraint involves roles from two predicates (`studentInvolvedI n` and `courseInvolvedIn`), so this is an external frequency constraint. To enforce this constraint, we first derive a function to compute the number of enrollments of any given student in any given course in which that student has enrolled:

```
nrEnrollmentsFor[s, c] = n -> Student(s), Course(c),
    int(n).
nrEnrollmentsFor[s, c] = n <-
```

```
    agg<<n = count()>> studentInvolvedIn[e] = s,
    courseInvolvedIn[e] = c.
// For each student s who enrolled in course c,
// n is the number of cases where there exists an
// enrollment e involving that s and that c.
```

You may wish to prepend the function name by "positive" to emphasize that function returns nothing (instead of 0) if s is not enrolled in c. For the data shown, querying the function nrEnrollmentsFor returns the following result:

```
120, CS100, 2
120, CS135, 1
501, CS135, 1
```

Now that this function is defined, we enforce the frequency constraint:

```
nrEnrollmentsFor[_, _] = n -> n <= 2.
// Each student enrolls at most twice in the same
// course.
```

If you attempt to violate this constraint (e.g., by adding a third enrollment of student 120 in CS100), an error is generated. The code for the above program and data, including a counterexample test, is available in the files Enrollment*.logic.

Subset and Exclusion Constraints Involving Join Paths

The conjunctive condition for the above nrEnrollmentsFor predicate uses the same enrollment variable e to equate the enrollment in both the studentInvolvedIn predicate and the courseInvolvedIn predicate. Such a matching of an argument in one atom with an argument in another atom is said to perform a **join** between the predicate roles involved.

A **join path** is a sequence of connected atoms, with each subsequent atom joined to the previous atom by matching an argument variable. For the purposes of this definition, we use the term *atom* liberally, to include functional applications such as fatherOf[p1]=p2.

Recall the following example of a subset constraint presented in Unit 2.4. This code expresses the constraint that stores with garden centers are a subset of stores that sell lawn mowers:

```
Store(s), hasStoreNr(s:n) -> int(n).
hasGardenCenter(s) -> Store(s).
sellsLawnMowers(s) -> Store(s).
```

```
hasGardenCenter(s) -> sellsLawnMowers(s).
// If store s has a garden center, then s sells
// lawn mowers.
```

Exclusion constraints also compare sets but require the constrained sets to be mutually exclusive. Here is an example from Unit 1.7. In this case, the exclusion constraint indicates that male monarchs and female monarchs form disjoint sets:

```
Monarch(m) ->. // Monarch is an entity type
// isMale(m) -> Monarch(m).
// If m is male then m is a monarch.
isFemale(m) -> Monarch(m).
// If m is female then m is a monarch.
isMale(m) -> !isFemale(m).
// If m is male then m is not female.
```

For a single role, the arguments of subset and exclusion constraints must be compatible with each other, and the relevant set-comparison operation (subset or exclusion) should be applied between them. Note that both kinds of constraints may also apply to compatible sequences of roles.

For any predicate, a subset of its arguments can be selected for comparison with other predicates. Such a subset is called a **projection**. Projections can also be performed on the set of predicates involved in a join path. If an argument of a subset or exclusion constraints is projected from a join path, it is called a *join-subset constraint* or *join-exclusion constraint,* respectively.

As an example of a join-subset constraint, recall the following constraint from the Consolidation Exercise in Chapter 2: If person `p` has a title `pt` that applies to only one gender `g`, then person `p` must be of gender `g`. Here the set of (`p, g`) pairs projected from the join path `personTitleOf[p]=pt, applicableGenderOf[pt]=g` is required to be a subset of the set of (`p, g`) pairs populating `genderOf[p]=g`:

```
personTitleOf[p] = pt, applicableGenderOf[pt] = g ->
    genderOf[p] = g.
// If person p has a person title pt that applies only
// to a specific gender g, then person p must be of
// gender g.
```

As an example of a join-exclusion constraint, consider the report shown in Table 4.12 from a paper-review system. Each person is affiliated with

TABLE 4.12 Reviewer Participation Data

Person	Institute	Papers Assigned	Papers Authored
Ann Jones	UCLA	20	10
Bill Smith	MIT		15
Cate Wong	UCLA	20	
Dan Green	MIT	10	15, 20
Eve Noon	UCLA	15	

exactly one institute, and each person authored a paper or is assigned to review a paper (or both).

A basic schema for this report is as follows:

```
Person(p), hasPersonName(p:n) -> string(n).
Institute(i), hasInstituteName(i:n) -> string(n).
Paper(p), hasPaperNr(p:n) -> int(n).
instituteOf[p] = i -> Person(p), Institute(i).
authored(p, ppr) -> Person(p), Paper(ppr).
isAssigned(p, ppr) -> Person(p), Paper(ppr).
Person(p) -> instituteOf[p] = _.
Person(p) -> isAssigned(p, _) ; authored(p, _).
```

To avoid potential bias in reviews, it is common practice to ensure that no person may review a paper authored by a person from the reviewer's institute. The current data satisfy this constraint but could easily be violated by another update (e.g., assign paper 10 to be reviewed also by Cate Wong).

Conceptually, this constraint involves projecting person-and-paper pairs from one join path (from institute to author to paper) and excluding these from the person-and-paper pairs projected from another join path (from institute to reviewer to paper). To facilitate the coding of this join-exclusion constraint, it helps to first derive predicates that project the person-and-paper pairs from the relevant join paths and then assert exclusion between these derived predicates:

```
hasAnAuthorOf(i, ppr) -> Institute(i), Paper(ppr).
hasAnAuthorOf(i, ppr) <- instituteOf[p] = i,
    authored(p, ppr).
hasAReviewerOf(i, ppr) -> Institute(i), Paper(ppr).
hasAReviewerOf(i, ppr) <- instituteOf[p] = i,
    isAssigned(p, ppr).
```

```
hasAnAuthorOf(i, ppr) -> !hasAReviewerOf(i, ppr).
// Nobody may review a paper authored by someone from
// the same institute.
```

If you attempt to violate this constraint (e.g., by assigning paper 10 to be reviewed also by Cate Wong), an error is generated. The code for the above program and data, as well as a counterexample test, is available in the files `JoinExclusion*`.

Tip: To join two predicates with a common argument type, use a conjunction with a common variable for the matching argument.

Exercise 2A: The report extract shown in Table 4.13 is from a weather bureau that records the minimum and maximum temperatures of some cities for the current day. Temperatures are measured in degrees Celsius. A basic program and data are given below and are also accessible as `CityTemp.logic`. Extend the program with code to enforce: (i) the equality constraint that for any given city, either both the temperatures are recorded or none are; and (ii) the value constraints that each minimum temperature is above –50°C, and each maximum temperature is in the range –10°C through 50°C (inclusive):

```
// Schema
City(c), hasCityName(c:n) -> string(n).
minCelsiusTempOf[c] = t -> City(c), int(t).
maxCelsiusTempOf[c] = t -> City(c), int(t).

// Data
+minCelsiusTempOf["Brisbane"] = 14.
+maxCelsiusTempOf["Brisbane"] = 27.
+minCelsiusTempOf["Sydney"] = 14.
+maxCelsiusTempOf["Sydney"] = 19.
+City("Melbourne").
```

TABLE 4.13 City Temperature Extremes

City	Minimum Temperature (°C)	Maximum Temperature (°C)
Brisbane	14	27
Melbourne		
Sydney	14	19

Exercise 2B: The report extract shown in Table 4.14 is from a weather bureau that records the minimum and maximum temperatures of some cities for some months of the current year. Temperatures are measured in degrees Celsius. A basic program and data are given below and are also accessible as `CityTemp2.logic`. Extend the program with code to enforce the equality constraint that for any given city and month, either both the temperatures are recorded or none are:

```
// Schema
City(c), hasCityName(c:n) -> string(n).
Month(m), hasMonthCode(m:c) -> string(c).
minCelsiusFor[c, m] = t -> City(c), Month(m), int(t).
maxCelsiusFor[c, m] = t -> City(c), Month(m), int(t).

// Data
+minCelsiusFor["Brisbane", "Jan"] = 20.
+maxCelsiusFor["Brisbane", "Jan"] = 36.
+minCelsiusFor["Brisbane", "Feb"] = 20.
+maxCelsiusFor["Brisbane", "Feb"] = 33.
+City("Melbourne").
```

Exercise 2C: In the report in Table 4.15, the second column records the languages in which the person on that row is fluent. Todd Ler is not yet fluent in any language (perhaps because he is just a baby!). The third and

TABLE 4.14 City Temperature Extremes per Month

City	Month	Minimum Temperature (°C)	Maximum Temperature (°C)
Brisbane	January	20	36
Brisbane	February	20	33
Melbourne			

TABLE 4.15 Languages Mastered

Person	Languages Mastered	Is Bilingual?	Is Translator?
Ann Jones	English	No	No
Bill Smith	English, Spanish	Yes	Yes
Cate Banderas	French, Spanish	Yes	No
Fumie Kano	English, Japanese, Mandarin	Yes	Yes
Todd Ler		No	No

fourth columns indicate whether or not that person is bilingual (i.e., fluent in at least two languages) or employed as a translator, respectively.

A basic program and data are given below and are also accessible as `Languages.lb`. (i) Extend the program using the `count` function to derive the positive number of languages in which a person is fluent. Then use that function to derive whether a person is bilingual and enforce the constraint that each translator is bilingual. (ii) Without using the `count` function, write a derivation rule to determine whether a person is bilingual. Would it be convenient to use this approach for larger frequencies (e.g., to find those people who speak at least five languages)?

```
Person(p), hasPersonName(p:pn) -> string(pn).
Language(la), hasLanguageName(la:ln) -> string(ln).
isFluentIn(p, la) -> Person(p), Language(la).
Translator(t) -> Person(t).
lang:isEntity[`Translator] = true.
```

The schema may be populated with the data shown by executing these updates:

```
+isFluentIn("Ann Jones", "English").
+isFluentIn("Bill Smith", "English"),
    +isFluentIn("Bill Smith", "Spanish"),
    +Translator("Bill Smith").
+isFluentIn("Cate Banderas", "French"),
    +isFluentIn("Cate Banderas", "Spanish").
+isFluentIn("Fumie Kano", "English"),
    +isFluentIn("Fumie Kano", "Japanese"),
    +isFluentIn("Fumie Kano", "Mandarin"),
    +Translator("Fumie Kano").
+Person("Todd Ler").
```

Exercise 2D: The report extracts in Tables 4.16 and 4.17 indicate official languages of some countries (BE, Belgium; CA, Canada; FI, Finland), as well as the languages mastered by ambassadors to those countries.

TABLE 4.16 Official Languages for Countries

Country	Official Languages
Belgium (BE)	Dutch, French, German
Canada (CA)	English, French
Finland (FI)	Finnish, Swedish

TABLE 4.17 Ambassadors and Their Languages

Person	Languages Mastered	Ambassador to Country
Ann Jones	English	Canada (CA)
Bob Adams	English, Finnish	Finland (FI)
Cate Brown	Dutch, French, Swedish	Belgium (BE)

A basic program and data are provided below and are accessible as `Ambassador.lb`. We now wish to ensure that if a person is an ambassador to a country then that person must be fluent in at least one official language of that country. Add code to enforce this join-subset constraint. *Hint*: First derive a predicate to project `p` and `c` from the join path:

```
// Schema
Person(p), hasPersonName(p:n) -> string(n).
Language(la), hasLanguageName(la:n) -> string(n).
Country(c), hasCountryCode(c:cc) -> string(cc).
isFluentIn(p, la) -> Person(p), Language(la).
hasOfficialLanguage(c, la) -> Country(c), Language(la).
countryAmbassadoredBy[p] = c -> Person(p), Country(c).

// Data
+hasOfficialLanguage("BE", "Dutch"),
    +hasOfficialLanguage("BE", "French"),
    +hasOfficialLanguage("BE", "German").
+hasOfficialLanguage("CA", "English"),
    +hasOfficialLanguage("CA", "French").
+hasOfficialLanguage("FI", "Finnish"),
    +hasOfficialLanguage("FI", "Swedish").
+isFluentIn("Ann Jones", "English").
+isFluentIn("Bob Adams", "English"),
    +isFluentIn("Bob Adams", "Finnish").
+isFluentIn("Cate Brown", "Dutch"),
    +isFluentIn("Cate Brown", "French"),
    +isFluentIn("Cate Brown", "Swedish").
+countryAmbassadoredBy["Ann Jones"] = "CA".
+countryAmbassadoredBy["Bob Adams"] = "FI".
+countryAmbassadoredBy["Cate Brown"] = "BE".
```

Exercise 2E: In the report extract shown in Table 4.18, products are identified by name, employees by employee numbers (e1, e2, etc.), and reviews by review numbers. A product may have many developers and

TABLE 4.18 Product Review Data

Product Name	Product Developers	Review Number	Reviewer
WordLight	e1, e2	1	e3
		2	e4
FunBlox	e1	3	e2

many reviews, but each review is for exactly one product and is authored by exactly one employee.

A basic program and data for this report are shown below and are also accessible in `ProductReview.lb`. Extend the code by adding a join-exclusion constraint to ensure that no employee may review a product for which he/she was a developer:

```
// Schema
Employee(e), hasEmployeeNr(e:n) -> string(n).
Product(p), hasProductName(p:n) -> string(n).
Review(r), hasReviewNr(r:n) -> int(n).
developed(e, p) -> Employee(e), Product(p).
authorOf[r] = e -> Review(r), Employee(e).
productReviewedIn[r] = p -> Review(r), Product(p).
Review(r) -> authorOf[r] = _, productReviewedIn[r] = _.

// Data
+developed("e1", "Wordlight"), +developed("e1",
    "FunBlox").
+developed("e2", "Wordlight").
+authorOf[1] = "e3", +productReviewedIn[1] =
    "Wordlight".
+authorOf[2] = "e4", +productReviewedIn[2] =
    "Wordlight".
+authorOf[3] = "e2", +productReviewedIn[3] = "FunBlox".
```

UNIT 4.3: DERIVED ENTITIES AND CONSTRUCTORS

This unit discusses a way to add information to the extensional database (EDB) that complements the use of delta logic that we have seen previously. In particular, it allows us to specify predicates that entail the existence of new entities and have the LogiQL execution engine add those entities to the database. These added entities are called **derived entities**, and the predicates used to define them are called **constructor** predicates.

Recall that a derivation rule is used to derive new facts from facts that have already been asserted or derived. For example, the following rule may be used to derive the fact that person `p1` is an uncle of person `p2`, given that we already know that `p1` is a brother of some person `p3` who is a parent of `p2`:

```
isUncleOf1(p1, p2) -> Person(p1), Person(p2).
isUncleOf1(p1, p2) <- isBrotherOf(p1, p3),
    isParentOf(p3, p2).
```

Recall also that a delta rule with delta modifiers in its head is used to update the extensional database when triggered by an update to a predicate in the body of the rule. For example, the following delta rule could be used to add relevant unclehood facts into the EDB when the relevant brotherhood and parenthood facts are added to the EDB:

```
isUncleOf2(p1, p2) -> Person(p1), Person(p2).
+isUncleOf2(p1, p2) <- +isBrotherOf(p1, p3),
     +isParentOf(p3, p2).
```

Except for the delta modifiers, this delta rule looks similar to the derivation rule above it. However, it behaves quite differently. Delta rules are triggered by update events noted in their bodies and are fully evaluated in the initial stage of the transaction. In contrast, derivation rules without delta modifiers are evaluated incrementally and are executed in the final stage of the transaction. Moreover, if some facts involved in a rule body are updated in a later transaction, the LogiQL execution engine automatically takes appropriate action.

For example, if a brotherhood or parenthood fact used by the derivation rule to derive an `isUncleOf1` fact is subsequently retracted and there are no other brotherhood and parenthood facts that satisfy the condition in the rule body for that unclehood fact, then the derived `isUncleOf1` fact is automatically deleted. If instead you use the above delta rule, then you would have to manually write the code to perform compensating updates in response to future updates. Since the maintenance of data updates for derivation rules is automatically managed by the LogiQL execution engine, such rules may often provide a better coding alternative than delta rules that include updates in their heads.

One case where delta rules are useful is when dealing with pulse predicates. Recall the following example from the previous chapter, where a

delta rule is used to add an employee to the EDB with the relevant details entered on a screen form:

```
+Employee(e), +hasEmpNr(e:n), +familyNameOf[e] = fn,
    +givenNameOf[e] = gn <-
    +okButtonIsPressedOn(f), +textField:empNrOn[f] = n,
    +textField:familyNameOn[f] = fn,
    +textField:givenNameOn[f] = gn.
```

Note that the updates in the head of this delta rule include a variable `e` that does not occur in the body of the rule. This is allowed because the `Employee` predicate is preceded by a delta modifier. If no delta modifier is included, we earlier forbade this situation, because our first safety condition for derivation rules requires each head variable (in a non-delta predicate) to occur in a positive context in the rule body. However, as an extension beyond classical Datalog, LogiQL allows this safety condition to be overridden under the special circumstances described in this unit, thereby enabling a rule to derive the existence of a new entity.

As a simple example to introduce the basic concepts and syntax, consider the report shown in Table 4.19 about countries and their presidents (as of 2011). Here, countries are identified by their country code (DE, Germany; FI, Finland; FR, France; IE, Ireland; US, United States). Note that for this application domain, the names of the presidents are not of interest.

One way of expressing this example in LogiQL is shown below. For simplicity, years are represented simply as integers:

```
// Schema
Country(c), hasCountryCode(c:cc) -> string(cc).
Gender(g), hasGenderCode(g:gc) -> string(gc).
genderOfPresidentOf[c] = g -> Country(c), Gender(g).
birthyearOfPresidentOf[c] = y -> Country(c), int(y).
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".

// Data
+genderOfPresidentOf["DE"] = "M",
   +birthyearOfPresidentOf["DE"] = 1959.
+genderOfPresidentOf["FI"] = "F",
   +birthyearOfPresidentOf["FI"] = 1943.
+genderOfPresidentOf["FR"] = "M",
   +birthyearOfPresidentOf["FR"] = 1955.
```

TABLE 4.19 Gender and Birth Years of Presidents

Country	President's Gender	President's Birth Year
DE	M	1959
FI	F	1943
FR	M	1955
IE	F	1951
US	M	1961

Note: DE, Germany; FI, Finland; FR, France; IE, Ireland; US, United States.

```
+genderOfPresidentOf["IE"] = "F",
    +birthyearOfPresidentOf["IE"] = 1951.
+genderOfPresidentOf["US"] = "M",
    +birthyearOfPresidentOf["US"] = 1961.
```

The data in the report can be retrieved by running the following query, which returns the result shown:

```
_(c, g, y) <-
   genderOfPresidentOf[c] = g,
       birthyearOfPresidentOf[c] = y.
```

Result:

```
US, M, 1961
IE, F, 1951
FR, M, 1955
FI, F, 1943
DE, M, 1959
```

The code is available as `President1.lb`.

Although this approach works, it seems somewhat unnatural to treat the gender and birth details directly as facts about countries, using the `genderOfPresidentOf` and `birthyearOfPresidentOf` predicates. It seems cleaner to think of the gender and birth year details directly as facts about the presidents of those countries (and hence only indirectly as facts about countries). For example, we might informally verbalize the last row of the report thus: "The president of Germany is male and was born in 1959." In so doing, we are explicitly mentioning presidents as an entity type of interest. Moreover, we are using a definite description, "the president of Germany," to identity one entity (the German president) by its presidential relationship to another entity (the country Germany). This is an example of an *entity-to-entity* reference scheme.

In contrast, refmode predicates identify an entity by relating it to a data value, and hence provide an *entity-to-value* reference scheme. For example, the refmode declaration for `Country` enables us to use "`Country("DE")`" as shorthand for the definite description "the country that has the country code 'DE.'" In LogiQL, if an entity type has no associated refmode predicate (is *refmodeless*), you can provide a reference scheme for it by declaring at least one constructor to determine instances of it.

The following lines of code show how to provide such a reference scheme for the entity type `President` in our current example:

```
President(p) ->.
// President is a refmodeless entity type
presidentOf[c] = p -> Country(c), President(p).
lang:constructor(`presidentOf).
// presidentOf is a constructor that maps countries to
// their presidents.
```

The first line declares `President` as an entity type without a refmode. The next two lines of code declare the functional predicate `presidentOf`, using the metapredicate `lang:constructor` to indicate that `presidentOf` is a constructor. This means that the function application `presidentOf[c]` provides a primary way to reference the president of country `c` and that the `presidentOf` predicate is 1:1.

If a rule containing a constructor in its head executes, one of two things happens, depending on the value of the constructor's key. If the constructor predicate does not already contain a fact with that key, then a new entity is created as the constructor's value, and the constructor predicate is updated to contain a new fact with the given key and new entity. If the constructor does contain a fact with the given key then the already associated value is reused. Hence, there is no need to explicitly code a uniqueness constraint to ensure that a constructor is inverse functional.

In our president example, the following derivation rule may now be used to derive the existence of a president and the presidential relationship to his or her country from the mere fact that its country exists. Note also that the head of the rule in this example is a conjunction:

```
President(p), presidentOf[c] = p <- Country(c).
// For each country there exists a president
// who is the president of that country only.
```

Understanding constructor predicates depends on the concept of *logical quantification*, which is more fully described in Appendix G to this book. Thus far, the variables we have seen in the heads of derivation rules have been *universally quantified*. What this means is that the rules should hold true for all possible values of those variables. In contrast, variables that occur only in the body of a rule are *existentially quantified*. This means that the rule body is satisfied if it holds for at least one value of the variable. For example, the unclehood derivation rule considered above means: *for each* person `p1` and *for each* person `p2`, `p1` is an uncle of `p2` if *there exists* some person `p3` such that `p1` is a brother of `p3` and `p3` is a parent of `p2`.

However, because `President` is a refmodeless entity type and has a constructor-based reference scheme, the presidential derivation rule above is interpreted differently from a normal derivation rule. The `President` variable `p` is allowed to appear only in the head of the rule and is assumed to be existentially quantified. Hence, the derivation rule may be read as follows: *For each country* `c`*, there exists a president* `p` *who is the president of* `c`. Because the individual variable `p` is existentially quantified in the head of the derivation rule, it is known as a **head existential**. An entity whose existence is inferred from such a derivation rule is called a *derived entity*.

Now that `President` is explicitly included as an entity type, we may introduce predicates to express gender and birth year details directly as facts about presidents. For example, the following rules may be used to derive these predicates from the predicates used in the original data entry:

```
genderOf[p] = g -> President(p), Gender(g).
genderOf[p] = g <- presidentOf[c] = p,
    genderOfPresidentOf[c] = g.
birthyearOf[p] = y -> President(p), int(y).
birthyearOf[p] = y <-
    presidentOf[c] = p, birthyearOfPresidentOf[c] = y.
```

One advantage of doing this is to enable rules or queries about presidents to be conceptualized and expressed in a more natural fashion that explicitly references presidents. For example, the above query to display the countries and the gender and birth year of their presidents may now be reformulated as follows:

```
_(c, g, y) <- presidentOf[c] = p, genderOf[p] = g,
    birthyearOf[p] = y.
```

The code for the above program, data, and query is available as `President2.lb`.

Note that its derived entities and derived predicates were not used to actually enter the data. In practice, this is often the best approach. However, it is possible to use derived entities and derived predicates to effectively enter data, as shown by the following code, available in `President3.lb`:

```
// Schema
Country(c), hasCountryCode(c:cc) -> string(cc).
Gender(g), hasGenderCode(g:gc) -> string(gc).
President(p) ->.
presidentOf[c] = p -> Country(c), President(p).
lang:constructor(`presidentOf).
President(p), presidentOf[c] = p <- Country(c).
genderOf[p] = g -> President(p), Gender(g).
birthyearOf[p] = y -> President(p), int(y).
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".

genderOf[p] = "M", birthyearOf[p] = 1959 <-
    presidentOf["DE"] = p.
genderOf[p] = "F", birthyearOf[p] = 1943 <-
    presidentOf["FI"] = p.
genderOf[p] = "M", birthyearOf[p] = 1955 <-
    presidentOf["FR"] = p.
genderOf[p] = "F", birthyearOf[p] = 1951 <-
    presidentOf["IE"] = p.
genderOf[p] = "M", birthyearOf[p] = 1961 <-
    presidentOf["US"] = p.
// Derive the gender and birthyear facts.

// Data
+Country("DE"), +Country("FI"), +Country("FR").
+Country("IE"), +Country("US").
+Gender("M"), +Gender("F"). // Needed for
    genderOf rule.
```

The predicates `genderOfPresidentOf` and `birthyearOfPresidentOf` are absent, and the only asserted facts are for the country and gender entities. The facts about the gender and birth years of presidents are now derived from the derivation rules shown. In declaring the `genderOf` function, we specified `Gender` (an entity type) as the type of its value parameter. Because of this, the derivation rule for `genderOf`

relies on the existence of the gender entities, so they must be explicitly added as shown in the final line of the data entry code. In contrast, the `birthyearOf` function returns a primitive value rather than an entity, so its derivation rule needs no prior declaration of the year numbers.

Because constructors provide a way to deal with unnamed, derived entities, they are useful for transforming one structure to a corresponding structure in a different notation. As an example, let's consider a simplified fragment of a program to generate English verbalizations of constraints that are captured in an information model or a LogiQL program. For instance, the functional nature of `genderOf[p]=g` involves a uniqueness constraint to ensure that each person has at most one gender. This hard constraint could be verbalized as follows: *It is necessary that each person has at most one gender*. Similarly, the soft uniqueness constraint that each person ought to have at most one wife could be verbalized as follows: *It is obligatory that each person is husband of at most one person*. In LogiQL, a soft constraint may be implemented by a rule to produce an error message when the constraint is violated. Let us name these hard and soft uniqueness constraints UC1 and UC2, respectively.

The hard or soft nature of a constraint is called its *modality*, and each verbalization starts with some modal text (e.g., "It is necessary that") to express the modality. The rest of the verbalization includes the names of the object types as well as phrases to capture relevant quantifiers and predicates. For simplicity, we consider here just the task of automatically generating the modal text for the constraint verbalization.

A verbalization itself has a textual form but does not have a name; hence, it is refmodeless and may be identified by a definite description such as "the verbalization of constraint UC1." Since we wish to generate the text of the verbalizations from the constraints, this task is an obvious candidate for the derived entity approach, as used in the following program:

```
// Schema
Constraint(c), hasConstraintName(c:cn) -> string(cn).
Modality(m), hasModalityName(m:mn) -> string(mn).
hasModalityName(_:mn) -> mn = "Hard" ; mn = "Soft".
modalityOf[c] = m -> Constraint(c), Modality(m).

ConstraintVerbalization(cv) ->.
verbalizationOf[c] = cv -> Constraint(c),
    ConstraintVerbalization(cv).
lang:constructor(`verbalizationOf).
```

```
modalTextOf[cv] = s -> ConstraintVerbalization(cv),
    string(s).
ConstraintVerbalization(cv), verbalizationOf[c] = cv,
    modalTextOf[cv] = "It is necessary that " <-
    modalityOf[c] = "Hard".
// If a constraint is of hard modality
// then there exists a verbalization of the constraint
// whose modal text reads "It is necessary that ".

ConstraintVerbalization(cv), verbalizationOf[c] = cv,
    modalTextOf[cv] = "It is obligatory that " <-
    modalityOf[c] = "Soft".
// If a constraint is of soft modality
// then there exists a verbalization of the constraint
// whose modal text reads "It is obligatory that ".

// Data
+modalityOf["UC1"] = "Hard".
+modalityOf["UC2"] = "Soft".
```

The following query may now be used to output the modal text parts of the constraint verbalizations, giving the results shown. The code for the program, data, and query is available as `Verbalize1.lb`:

```
_(c, mt) <-
    Constraint(c), verbalizationOf[c] = cv,
    modalTextOf[cv] = mt.
```

Result:

```
UC2, It is obligatory that
UC1, It is necessary that
```

Thus, head existentials provide a very powerful way to code data transformations, such as that in the previous example. They have also proven useful in other kinds of applications such as those involving data integration or view updates.

N-ary Constructors

The constructors considered so far have all been 1:1 binary predicates. It is also possible to have *n*-ary constructors ($n > 2$), where their first $n - 1$ roles provide the keyspace, and the value space also functionally determines the keyspace. To illustrate this, consider the report shown in Table 4.20

TABLE 4.20 Names and Birth Years of Presidents

President's Given Name, etc.	President's Family Name	Birth Year
Barack	Obama	1961
George W.	Bush	1946
Bill	Clinton	1946
George H. W.	Bush	1924
..	..	..

about presidents of the United States. Here, "President's Given Name, etc." includes a given name and optionally one or more initials to ensure that the combination of given name and family name suffices to identify a president.

A program schema describing the contents of this report, available as `President5.lb`, is as follows:

```
// Schema - Note: Incomplete without constructor
President(p) ->.
givenNameEtcOf[p] = gn -> President(p), string(gn).
familyNameOf[p] = fn -> President(p), string(fn).
birthyearOf[p] = y -> President(p), int(y).
President(p) -> givenNameEtcOf[p] = _,
    familyNameOf[p] = _.
// Each president has a given name and a family name.
givenNameEtcOf[p1] = gn, familyNameOf[p1] = fn,
givenNameEtcOf[p2] = gn, familyNameOf[p2] = fn ->
    p1 = p2.
// Each combination of GivenNameEtc and FamilyName
// applies to only one president.
```

In this example, the combination of the two names (given and family) provides a *composite reference scheme* for it. This reference scheme is enforced by the mandatory role and external uniqueness constraints shown above. Without these in place, it would be possible to add presidents with no names or with the same name combination as already added presidents. Note, however, that the above schema is not sufficient to describe the data because it does not offer a way to explicitly assert the data. That is, it provides neither a refmode nor a constructor.

It is possible to address the above concern by providing a refmode, but if none is naturally available, the developer must manufacture one.

It may be preferable, instead, to support such a reference scheme by using a ternary constructor to map each name combination to a `President` entity. For example, the following code uses the predicate `pairsWith` to enter the name pairs, and then uses the constructor function `presidentNamed[gn,fn]=p` to derive a president from that name pair. The constructor mapping is 1:1 between name pairs and presidents, so each president is assigned only one name pair:

```
// Schema
pairsWith(gn, fn) -> string(gn), string(fn).
birthyearOf[p] = y -> President(p), int(y).

President(p) ->.

presidentNamed[gn, fn] = p -> string(gn), string(fn),
    President(p).
lang:constructor(`presidentNamed).
// presidentNamed is a constructor
// that maps givenname, familyname pairs to presidents.

President(p), presidentNamed[gn, fn] = p <-
    pairsWith(gn, fn).
// For each paired combination of extended given name
// and family name, there exists a president
// who may be identified by that name combination.

// Data
+pairsWith("Barack", "Obama"),
    +pairsWith("George W.", "Bush"),
    +pairsWith("Bill", "Clinton"),
    +pairsWith("George H. W.", "Bush").
```

The birth year facts for the presidents may be derived as follows:

```
birthyearOf[p] = 1961 <-
    presidentNamed["Barack", "Obama"] = p.
birthyearOf[p] = 1946 <-
    presidentNamed["George W.", "Bush"] = p.
birthyearOf[p] = 1946 <-
    presidentNamed["Bill", "Clinton"] = p.
birthyearOf[p] = 1924 <-
    presidentNamed["George H. W.", "Bush"] = p.
```

The following query may now be used to display the presidential data:

```
_(p, gn, fn, by) <-
     presidentNamed[gn, fn] = p, birthyearOf[p] = by.
```

The program, data, and query are accessible in `President6.lb`.

Constructors may also be used to support *disjunctive reference schemes*, in which entities of a given type may be derived using any one of multiple constructors. In such cases, only one constructor may be used to derive any given instance of that type. For example, we might derive a person from either a Social Security Number or a passport number, but not both. The first question in the following exercise includes a further example of this kind.

Tip: Use constructors for deriving entities in order to provide a more natural way to conceive of some existing data or to transform an existing structure into another form.

Exercise 3A: The constraint verbalizations generated in `Verbalize1.lb` are examples of *positive* verbalizations. A constraint may also have a *negative* verbalization, indicating how to violate the constraint. For example, the hard constraint UC1 with the positive verbalization, *It is necessary that each person has at most one gender,* also has the negative verbalization, *It is impossible that some person has more than one gender*. Similarly, the soft uniqueness constraint UC2 that each person ought to have at most one wife is verbalized in positive form as *It is obligatory that each person is husband of at most one person*, and in negative form as *It is forbidden that some person is husband of more than one person*. For negative verbalizations, hard constraints have the modal text, *It is impossible that*, and soft constraints have the modal text, *It is forbidden that*. Modify the code in `Verbalize1.lb` to generate both positive and negative verbalizations. *Hint*: Replace the `verbalizationOf` constructor by two constructor predicates.

Exercise 3B: Consider the report shown in Table 4.21 about sports played by countries in international competitions. The third column includes the current rank of that country in that sport, if known. We could code this in LogiQL using a binary predicate to record which countries play which sports, and a ternary predicate to record their rank in that sport if known.

TABLE 4.21 Sports Rankings for Countries

Country	Sport	Rank
AU	Baseball	
AU	Basketball	3
AU	Cricket	4
US	Baseball	1
US	Basketball	1
US	Soccer	

Note: AU, Australia; US, United States.

A schema and data for this approach are shown below and are available in the file `Playing.lb`. The subset constraint ensures that countries are ranked only in sports that they have been asserted to play:

```
// Schema
Country(c), hasCountryCode(c:cc) -> string(cc).
Sport(s), hasSportName(s:sn) -> string(sn).
plays(c, s) -> Country(c), Sport(s).
sportRankFor[c, s] = r -> Country(c), Sport(s), int(r).
sportRankFor[c, s] = _ -> plays(c, s).
// If a country is ranked in a sport then it plays
// that sport.

// Data
+plays("AU", "baseball"), +plays("AU", "basketball"),
    +plays("AU", "cricket").
+sportRankFor["AU", "basketball"] = 3,
    +sportRankFor["AU", "cricket"] = 4.
+plays("US", "baseball"), +plays("US", "basketball"),
    +plays("US", "soccer").
+sportRankFor["US", "baseball"] = 1,
    +sportRankFor["US", "basketball"] = 1.
```

The second row of the report could be verbalized in two steps, starting with *Australia plays basketball*, and then saying *that playing is at rank 3*. Here the second sentence refers back to the previous sentence, using a noun phrase *that playing* to make an object out of the situation or state of affairs underlying the proposition captured by the first sentence. This linguistic nominalization process is common in data modeling, where it is known as *objectification* or *reification*.

Objectification may be coded in LogiQL by deriving the entity resulting from the nominalization and also deriving link predicates to relate it back to the original entities involved in its derivation.

Write code to extend the above program to derive `Playing(p)` from `plays` facts using the constructor `playingDerivedFrom[c,s]=p`. Then derive the link predicates `sportLinkedTo[p]=s` and `countryLinkedTo[p]=c` and the predicate `rankOf[p]=r` for `Playing(p)`.

UNIT 4.4: PROGRAM ORGANIZATION

The programs you have seen so far in this book have been intentionally quite modest in size so that they could focus on specific language features. Real programs, however, can include thousands of lines of code. Consequently, it is imperative to organize your programs in a way that makes them easy to understand, maintain, and reuse. Fortunately, LogiQL provides a rich array of mechanisms to support this need.

The general concept that you need to understand in order to properly structure your programs is *modularity*. A program is modular to the extent that dependencies among the various units of code that comprise the whole program are weak. If this is the case, the units are said to be *loosely coupled*. Conversely, if two units are *tightly coupled* then a change to one likely means that the other has to be changed, thereby increasing the effort required to make the change.

Another important aspect of modularity is *cohesion*. A unit of code is cohesive to the extent that it has a single purpose. If a unit of code is not cohesive, then it is likely difficult to understand and to reuse. With coupling and cohesion in mind, your goal in structuring your program code should be to break it up into modular units, which, individually, have a single purpose and, collectively, have as few interdependencies as possible.

Legacy Code

Most of the modularity features of LogiQL have recently been added to the language. Consequently, there is a lot of older, *legacy*, code in existence that does not make use of them. Instead, such code uses two informal mechanisms, which we describe here in case you have to maintain or reuse such code.

The first modularity mechanism that programmers used was to organize their code into files. LogiQL code loaded into a workspace from a `.logic` file is called a *block*. A block is nothing more than the internal

version of a file containing LogiQL code, and the only operations available on blocks are that they can be loaded into the current workspace and that they can be activated and deactivated.

The second means that programmers used to organize their legacy code was to employ naming conventions to group related predicates. In particular, members of a group of related predicates were given a common prefix, separated from the predicates' individual names with colons. Such a prefix provides an informal **namespace** useful when reading programs. You have already seen this technique used with, for example, the `date-time` predicates introduced in Chapter 1. In some legacy code, multiple prefixes are used to denote higher-level groupings.

These two mechanisms, while still available in LogiQL, have been superseded by more powerful features including *modules* and *separate compilation*. Besides direct support for modularity, the new features also improve compilation efficiency and provide you with more meaningful error messages.

Program Organization

If you are going to build a large LogiQL program, there are three related constructs you should understand: *projects, modules,* and *concrete blocks*. The highest-level construct is the **project**, which manifests as a directory containing a project description file and a collection of resources, such as legacy code files or modules. A project can correspond to a complete application or to a self-contained library resource that might be of use to other projects. For example, you might build your application by combining a project implementing the application's functionality with a pre-existing statistical-processing library project.

A **module** is also a structuring mechanism, but at a somewhat lower level of abstraction than a project. A module also manifests as a directory containing logic files or subdirectories or both. The directory structure of a module provides a possibly compound namespace for your predicates so that they can be referenced from other modules.

The code files in a module take the form of interdependent **concrete blocks**. In addition to LogiQL code, a concrete block employs special syntax that enables you to provide aliases for predicate names and to specify how the module's predicates can be accessed from other modules.

Projects

You define a project in LogiQL by giving it a name and creating a directory to hold its source code. That directory contains logic files, subdirectories,

and a project description file. Two kinds of logic files are allowed, both having the `.logic` suffix. Files of the first kind are the legacy source files mentioned above. Files of the second kind contain concrete blocks, as described below. Likewise, subdirectories (and their sub-subdirectories) are of two sorts—traditional directories containing legacy `.logic` files and modules, also described below.

In the *project description file* (sometimes called a *project manifest*), you describe how these pieces fit together. Each line (other than comment lines beginning with `//`) specifies a name and a descriptor, separated by a comma.

A name field can be simple, in which case it designates a `.logic` file in the top-level directory, or it can comprise a directory path ending in a `.logic` file.

Table 4.22 lists important descriptors and their intended uses.

Here is an example of a simple project description file. It, along with the other files used in the body of this unit, can be found in the subdirectory `example`:

```
example, projectname

// A legacy code block.
D.logic, active

// A legacy code block in a subdirectory.
subdir/F.logic, active

// Directory containing a module.
mod, module

// Legacy stored query.
G.logic, inactive
```

TABLE 4.22 Project Descriptors

Descriptor	Use
`projectname`	Name of the project
`active`	Named file contains legacy code to be installed in a workspace as an active block
`inactive`	Named file contains a stored query
`module`	Named directory contains a LogiQL module
`library`	Named library that has been constructed from a previously compiled project
`execute`	Named file contains code to be executed when the project is installed in the workspace; normally contains data assertions for initializing the workspace

If this project file is named `example.project`, then the directory for the project could contain the following contents:

`example.project`	project description file
`subdir`	subdirectory for legacy code
`subdir/F.logic`	legacy `.logic` program file
`D.logic`	legacy `.logic` program file
`mod`	module subdirectory
`mod/A`	namespace
`mod/A/A.logic`	concrete block
`mod/B`	namespace
`mod/B/B.logic`	concrete block
`mod/C.logic`	concrete block
`mod/E.logic`	concrete block
`G.logic`	legacy `.logic` stored query file

Note that you did not have to list the non-legacy `.logic` files in your project manifest. The LogiQL compiler is smart enough to figure this out for you. Also, subdirectories within a module now constitute namespaces.

The information in the project description file is used by the LogiQL compiler to control the compilation process and to provide access to project resources from other projects. In particular, the resources described in the file are processed in the order listed. If one of your legacy `.logic` files references another (legacy or non-legacy), the latter should appear above the former in the project description file.

If you create the above `example.project` file, the listed subdirectories and corresponding `.logic` files (which, for purposes of illustration, could be empty), you can build the project by running the following command from within the top-level directory:

```
lb compile project - out-dir out example.project
```

In the above command, `out` is a subdirectory that will contain the compiled binary files.

Once compilation is complete, you can install the resulting project into an existing workspace by invoking `lb` as follows:

```
lb addproject workspaceName directoryName
```

where *workspaceName* is the name of an existing workspace and *directoryName* is the name of directory containing your project file.

Alternatively, you can create the workspace during the same `lb` execution by adding the `create` option to the command line.

Modules and Concrete Blocks

LogiQL modules provide a disciplined way to manage names and to provide access to resources. This should be contrasted to the informal way in which names are managed in legacy programs.

Using modules, access to predicates defined in one module by predicates defined inside other modules can be controlled by the programmer. For example, if you know that a particular predicate is of use only in supporting a computation internal to a module, then you can prevent its alteration from within rules in other modules by *sealing* it. In this way, if you later wish to change its implementation, you reduce the work required to update the other modules.

Conversely, if you wish to treat a predicate as a resource for use by other modules, you can explicitly *export* it. Similarly, another module can access a predicate defined in your module, possibly giving it an alternative name (an *alias*) to avoid naming conflicts or to better express its responsibility in the importing module.

A module takes the form of a directory containing special (non-legacy) logic files (concrete blocks) and subdirectories (namespaces). The reason that the logic files are special is that they contain instructions to the compiler concerning the way in which the predicates in that file can be accessed. Here is an example of one such file, named `B.logic`, found in the `mod` subdirectory:

```
block(`B) {
    export(`{
      r[x] = y -> mod:A:A:p(x), int(y).
      M(x), hasMid(x:i) -> int(i).
    }),
sealed(`{
        q(x, y) -> M(x), M(y).
    }),
alias(`mod:C:p, `otherp),
    clauses(`{
        p(x) -> int(x).
p(x) <- otherp(x).
        q(x, y) <- hasMid(x), hasMid(y), x < y.
    })
} <— .
```

The `block` predicate is a built-in predicate used to indicate that its argument predicate (`B`) is a concrete block. Note that the name of the predicate must match the name of the file containing this code (ignoring the `.logic` suffix). Within the braces are several kinds of instructions. The first instruction (`export`) indicates that any predicates declared in this section of the block, such as the functional predicate `r`, the entity type `M`, and its refmode `hasMid`, should be made available to other modules. Note also that `r` refers to predicate `p` defined in concrete block `A`. The `sealed` instruction is a variant of `export`, the difference being that although the named predicate `q` is visible outside of module `B`, its contents cannot be added to or changed externally.

The next instruction, `alias`, indicates that predicate `p` from concrete block `C` can be referred to using the name `otherp`. This is useful to avoid confusion with the separate predicate, also named `p`, referred to in the `export` instruction.

The final instruction, `clauses`, defines yet another predicate named `p`. This one can be altered only within the current concrete block. It also contains a rule defining how the values of `q` are derived.

In summary, the LogiQL concrete block described above refers to three predicates named `p`. The first belongs to concrete block `A` and is referred to using its full name `mod:A:A:p`. The second `p` comes from a different concrete block `C` and is accessed through the alias `otherp`. The third predicate `p` can only be accessed from within this source file and is referred to by the unadorned name `p`.

Any logic files wishing to take advantage of the power of modules need to use these instructions. Non-module legacy logic files can still be included in a project by listing them in the project description file.

Namespaces Revisited

In the above example, the name "`mod:A:A:p`" was used to refer to predicate `p` defined in concrete block `A`. As the syntax suggests, there is a hierarchy of names, where each level corresponds to a directory or file within the module. Because modules are organized using directories, you can easily manage predicate names by building an appropriate directory hierarchy. In abstract terms, the syntax of names used in concrete blocks is the following, where square brackets ('`[]`') are used to surround optional constructs, and the asterisk ('`*`') is used to indicate any number of occurrences:

`[[`*moduleName*`[:`*directoryName*`]*:]`*fileName*`:]`*predicateName*

That is, a predicate name may be preceded by the name of the non-legacy `.logic` file containing it (without the `.logic` suffix) and by a sequence of directory names beginning with the name of the module. All names are separated by colons (':'). The beginning point of the sequence is the directory containing the module. Note that legacy predicates can be named directly, without any prefixes.

Each module describes a scheme for naming resources (predicates, concrete blocks, and namespaces). This is similar to the naming scheme described above for legacy code. However, the difference is that, here, the naming scheme is enforced by the compiler.

Separate Compilation and Libraries

Using modules has another benefit in addition to those described above—they make use of LogiQL's *separate compilation* facility. Previously, logic files were compiled when they were added to a workspace. A disadvantage of doing this is that unchanged code has to be recompiled if the workspace is reconstructed. With separate compilation, the redundant recompilation can be avoided by saving the results of the compilation in an intermediate file.

The result of compiling the file `a.logic` is the file `a.lbb`, containing a condensed form of the program that can be efficiently added to a workspace. A file named `LB_SUMMARY.lbp` is also produced to summarize the results of compiling the entire project. On subsequent executions, the LogiQL compiler uses these files to determine exactly what needs to be recompiled.

The result of compiling a project is called a **library**, and libraries can be used much as they can in other languages to hold pre-compiled collections of related resources. For example, the locations of library resources can be encoded in the `LB_LIBRARY_PATH` environment variable for use by the compiler.

The code for the above example (project file, directory hierarchy, and example logic files) can be found in the `example` subdirectory.

Summary

Table 4.23 summarizes the structuring features introduced in this unit.

Tip: Organize your code into modules that exhibit high internal cohesion and low coupling with other modules.

TABLE 4.23 LogiQL Structuring Features

Concept	Realization	Contents	Features
`project`	Directory and project description file	Legacy and non-legacy code files	Legacy code inclusion, executed assertions, references to external libraries
`module`	Directory	Submodules and concrete blocks	Naming scheme
`concrete block`	`.logic` file	Exports, seals, aliases, level-0 code	Controlled recompilation, aliasing, sealing
`library`	`.lbp` file	Rules, declarations, constraints, facts	Resource for use by other projects without recompilation
`export`	Metapredicate assertion	Set of declarations	Access to predicates
`alias`	Metapredicate assertion	Pairs of predicate names	More appropriate names and avoidance of name collision
`seal`	Metapredicate assertion	Set of declarations	Protection from change

Tip: Use an alias when an imported predicate uses the same name as one in your module.

Tip: Export only those predicates which you explicitly wish other modules to make use of.

Exercise 4A: You are hosting a dinner party to be attended by nine of your friends. You wish to arrange seating for the invitees among the three, three-person tables available so that people who already know each other are seated together as much as possible. Here are a schema and data describing the situation:

```
// Schema
Person(p), hasPersonName(p:s) -> string(s).
isFriendsWith(p1, p2) -> Person(p1), Person(p2).
Table(t), hasTableNbr(t:n) -> int(n).
sitsAtTable[p] = t -> Person(p), Table(t).

// Data
+Table(1).
+Table(2).
+Table(3).
```

```
+isFriendsWith("Snoopy", "Spike").
+isFriendsWith("Snoopy", "Marbles").
+isFriendsWith("Snoopy", "Olaf").
+isFriendsWith("Snoopy", "Belle").
+isFriendsWith("Spike", "Belle").
+isFriendsWith("Marbles", "Andy").
+isFriendsWith("Marbles", "Olaf").
+isFriendsWith("World War I Flying Ace", "Olaf").
+isFriendsWith("World War I Flying Ace", "Andy").
+isFriendsWith("Andy", "Missy").
+isFriendsWith("Belle", "Missy").
+isFriendsWith("Missy", "Joe Cool").
```

How would you arrange the tables for your party? *Notes:* There may be more than one best answer; you do not have to worry about seating yourself; you do not have to write any code, just list the seating arrangement.

Exercise 4B: In the Consolidation Exercise in Chapter 1, you built a comprehensive solution to computing various statistics about monarchs, such as their ages and the lengths of their reigns. The solution provided comprised a set of unstructured `.logic` files. You are now asked to organize your solution and the accompanying schema and data files using the program-structuring mechanisms introduced in this unit.

You can find the `.logic` files to use for this exercise in the directory `exercise/original`. Also present is a script file useful for querying the defined predicates (`queryTest.lb`) and one for loading in the data (`dataLoad.lb`). Here is a list of the `.logic` file names, in alphabetical order (ignoring case):

```
base.logic
baseData.logic
birthCountryData.logic
birthDeath.logic
birthDeathData.logic
Q1Query.logic
Q7Query.logic
Q8Query.logic
Q9Query.logic
```

(i) For the first part of this exercise, you should treat these files as legacy LogiQL files and construct a project description file suitable for use in compiling them. *Hint 1:* You need to reorder these entries to ensure

that predicates are compiled before they are referenced. *Hint 2*: You can ignore the data files; you will have to add them manually using the `lb` command. *Hint 3*: Use the `inactive` descriptor to describe the files containing queries. Test your solution by installing the project, asserting the facts in the data files, and running the four queries to see that you get the same results you did in Chapter 1.

(ii) The solution presented in part (i) has several problems: Some files provide predicates not closely related to each other (i.e., the files are not cohesive), some files are too tightly coupled with each other, there is no use of the features provided by concrete blocks, and, of course, the filenames are not clear indicators of the resources the named files provide. As a first step to addressing these problems, you are asked to construct a new module (in your project directory) called `chapter1CE` (for "Chapter 1 Consolidation Exercise") containing only a single file, called `Country.logic`. Remember that you will need to list the new module in your project description file.

`Country.logic` should be a concrete block that contains those parts of the other legacy files that pertain to countries. *Hint:* You will need to change legacy files that reference the predicates in `Country.logic` so that they now include a `Country:` prefix.

Make sure that your test queries still produce the same results.

(iii) A more ambitious task is to extract the code having to do with the dates of births, deaths, and reigns. Create a new concrete block in the `chapter1CE` module and call it `Date`. Place into it all of the code in the legacy files associated with these dates. Update the remaining legacy logic files by adding the appropriate prefix. Then compile, install, populate, and test your solution.

(iv) The solution for part (iii), while correct, still exhibits some problems that need to be fixed. One example occurs in the legacy file `Q9Query.logic`, which contains the following stored query:

```
_(m) <- Date:isFirst(m).
```

If you look in `Date.logic`, you will notice that `isFirst` is not exported. This limits its access from other non-legacy files, but not from legacy files. In anticipation of incorporating `Q9Query.logic`

into the `chapter1CE` module, examine `Date.logic` and place a declaration for `isFirst` into the `export` section. Also, determine which of the other predicates in the `clauses` section might be useful externally and place declarations for them into the `export` section as well. *Hint:* Only declarations may appear within the `export` section. Hence, you may have to add in some explicit declarations for the predicates you wish to export.

(v) Of the remaining legacy files, four correspond to queries and three to data. The other one—`base.logic`—resulted from collecting the LogiQL files used in Chapter 1. That is, it was not designed to be modular. In particular, it is not cohesive. For this exercise, you should refactor the resources provided by this file into at least three pieces and include them in the `chapter1CE` module.

(vi) Although there is no way in the current version of LogiQL to include data files within modules, query files can be included. To do this, you need to make several changes to the existing files: (1) Move the file containing the query into the module; (2) convert it into a concrete block, placing the query itself in the `clauses` section; and (3) change the query name so it is no longer anonymous. You can also use this opportunity to give the query file/block a more appropriate name. To try this out, take the query found in `Q9Query.logic` and convert it into a query named `FirstMonarchQuery.logic`.

UNIT 4.5: MISCELLANEOUS TOPICS

This unit contains a discussion of three topics that are too small to warrant units of their own. The first has to do with whether predicates are actually stored (*materialized*) in a workspace or are merely computed on demand. Depending on the circumstances, this choice can improve program efficiency. The second topic is LogiQL's hierarchical syntax, which is a shorthand for expressing the contents of a set of related predicates. The final topic concerns file predicates, which are one of LogiQL's ways of performing input from and output to your computer's file system.

Materialized and Derived-Only Views

The LogiQL execution engine normally processes rules that compute updates to a workspace. These derived predicates, whose fact instances are stored in this way for later use, are known as **materialized views**.

For many applications, materialized views provide the best option for realizing derivation rules, because the execution engine's incremental evaluation is often much more efficient than full re-evaluation. For example, consider a derivation rule to compute the current balance of a bank account. By storing the previous balance, when an account transaction (e.g., a deposit or withdrawal) occurs, we need to compute only the effect of that transaction on the account. Alternatively, if we do not materialize the balance, we would need to process all the past transactions on the account every time the account is updated, which could be quite an expensive operation.

However, it is sometimes useful to fully evaluate a derived predicate on the fly, making its result available for other rules within the transaction but not installing its result in the database when the transaction is committed. Such predicates are known as **derived-only predicates**, and using them requires that the compiler must be informed using the `lang:derivationType` metapredicate. The syntax is as follows, where *predicate* is the name of the derived-only predicate. The default setting for a predicate's derivation type is `DerivedAndStored`, which is the materialized-view case discussed above:

```
lang:derivationType[`predicate] = "Derived".
```

One reason for using derived-only predicates is to define a complex computation, which could potentially result in an infinite set of facts. Such a predicate can prove useful if we intend to use the result of the computation in a context that finitely constrains its arguments. That is, the full predicate need never be computed and stored. For example, the following program (available in the file `energy.lb`) uses the derived-only predicate `cSquared` to compute the energy equivalent (in Joules) of a given mass (in kilograms), by using Einstein's famous equation $E = mc^2$:

```
Mass(m), hasKgValue(m:kg) -> float(kg).
cSquared[kg] = j -> float(kg), float(j).
cSquared[kg] = j <- kg * pow[300000000f, 2f] = j.
lang:derivationType[`cSquared] = "Derived".

isHighEnergySource(m) -> Mass(m).
isHighEnergySource(m) <-
    Mass(m), hasKgValue(m:kg),
    cSquared[kg] = j, j > pow[10f, 18f].
```

```
isVeryHighEnergySource(m) -> Mass(m).
isVeryHighEnergySource(m) <-
    Mass(m), hasKgValue(m:kg),
    cSquared[kg] = j, j > pow[10f, 19f].
```

Here, *c* (the speed of light) is 3×10^8 meters/second, which we represent as `300000000f`. To square *c*, we raise *c* to the power of 2 using the built-in predicate `pow[base,power]=result`.

Two derivation rules are included to classify a mass as a *high-energy source* if its energy equivalent exceeds 10^{18} Joules, or as a *very high-energy source* if its energy equivalent exceeds 10^{19} Joules. Note that in the above code `cSquared` is designated as `Derived`. If, instead, it was treated as a derived-and-stored predicate, its rule would be unsafe. Can you see why?

First, of course, is the realization that storing this predicate is impractical. If fully computed, it would contain all pairs of floating-point numbers related by Einstein's formula. Moreover, recall from Chapter 3 that in order for a rule to be safe, each variable in the head of a rule (`m` in this case) must appear in a positive context as an argument to a domain predicate or a domain equality in the body of the rule, neither of which is the case here. Derived-only predicates, however, are treated specially by the execution engine. Their values are computed only in the context of the other rules that use them. In our example, `cSquared` is used inside of `isHighEnergySource` and `isVeryHighEnergySource`, both of which contain the atom `Mass(m)`, which uses a domain predicate. It is as if the text of the body of `cSquared` were literally copied into these two rules, thereby overcoming the safety objection.

We can assert the following facts to test our computation:

```
+Mass(10f).
+Mass(100f).
+Mass(1000f).
```

If we now query the computed predicates, the output is as shown below. Note that if you tried to print the population of the `cSquared` predicate, however, no facts would be displayed, because derived-only predicates do not have their results stored in the workspace:

```
Querying isHighEnergySource:
    1000
    100
Querying isVeryHighEnergySource:
    1000
```

Derived-only predicates can be useful in special circumstances, such as those mentioned above. However, there are several limitations in the rules used to compute them. In particular, the rules may not use negation and may not be disjunctive. The latter restriction means that they may not use the *or* operator and may not be split into several separate rules.

Hierarchical Syntax

A common practice in implementing an application is to begin with entities, and for each, define a set of property predicates expressing the data to be associated with those entities. For example, we might wish to include people in our application, and for each person, provide information about that person's name and address. Such a schema might look like the following:

```
Person(p), hasPersonNr(p:n) -> int(n).
firstNameOf[p] = s -> Person(p), string(s).
lastNameOf[p] = s -> Person(p), string(s).
streetAddressOf[p] = s -> Person(p), string(s).
cityOf[p] = s -> Person(p), string(s).
```

If we then wish to populate these predicates for a particular person, we could enter the data using delta predicates, as follows:

```
+Person(p),
    +hasPersonNr(p:13),
    +firstNameOf[p] = "John",
    +lastNameOf[p] = "Doe",
    +streetAddressOf[p] = "1384 West Peachtree Street",
    +cityOf[p] = "Atlanta".
```

Note that there is some duplication in the above, with `p` being referred to six times. This repetition would be even more irksome if we were entering data for many people. Fortunately, LogiQL has a shortcut that can reduce some of this burden. It is called **hierarchical syntax**. Here is how the above delta logic would be expressed with it:

```
+Person(p) {
    +hasPersonNr(13),
    +firstNameOf[] = "John",
    +lastNameOf[] = "Doe",
    +streetAddressOf[] = "1384 West Peachtree Street",
    +cityOf[] = "Atlanta"
}.
```

In the above, the curly braces ('{}') serve to enclose a set of atoms or function assignments, each of which is qualified by the particular entity that they describe. Note that the refmode predicate `hasPersonNr` did not have to include a colon. Moreover, because `p` was used only once, we did not actually have to include it. Instead, we could have replaced it with an underscore.

A hierarchical rule is similar to a normal logic rule, containing both a head and a body. However, instead of separating the two parts with an arrow, the body is enclosed in curly braces.

It gets even better—hierarchical rules can be nested. If we have a schema that warrants it, we can avoid even more repetition by including one set of fact assertions inside of another, as the following example illustrates:

```
// Schema
Person(p), hasPersonNr(p:n) -> int(n).
firstNameOf[p] = s -> Person(p), string(s).
lastNameOf[p] = s -> Person(p), string(s).
Address(a), hasStreetLocation(a:s) -> string(s).
addressOf[p] = a -> Person(p), Address(a).
cityIn[a] = s -> Address(a), string(s).

// Data
+Person(p),
    +hasPersonNr(p:13),
    +firstNameOf[p] = "John",
    +lastNameOf[p] = "Doe",
    +Address(a),
    +hasStreetLocation(a:"1384 West Peachtree
        Street"),
    +addressOf[p] = a,
    +cityIn[a] = "Atlanta".

// Hierarchically expressed data
+Person(_) {
    +hasPersonNr(14),
    +firstNameOf[] = "Jane",
    +lastNameOf[] = "Doe",
    +addressOf[] = +Address(_) {
       +hasStreetLocation("1384 West Peachtree Street"),
       +cityIn[] = "Atlanta"
    }
}.
```

There is one other feature of hierarchical syntax that you should be aware of—multiple entity types can be represented in the head. For example, have a look at the following schema:

```
Person(p), hasPersonName(p:s) -> string(s).
Car(c), hasBrand(c:s) -> string(s).
drives(p, c) -> Person(p), Car(c).
```

Without hierarchical syntax, we might assert facts to these predicates as follows:

```
+Person(p),
    +hasPersonName(p:"McCambridge"),
    +Car(c),
    +hasBrand(c:"Mercedes"),
    +drives(p, c).
```

Using hierarchical syntax, however, we can elide the arguments in the body and somewhat shorten the rule as follows:

```
(+Person(p), +Car(c)) {
    +hasPersonName[] = "McCambridge",
    +hasBrand[] = "Mercedes",
    +drives()
}.
```

Note that the head of the rule now contains two atoms, enclosed within parentheses. Despite the potential for ambiguity, the LogiQL compiler is able to determine which entity types belong with which predicates within the body.

Hierarchical syntax provides no additional power to the LogiQL language. Anything that can be expressed with it can be expressed without it. That being said, using hierarchical syntax can reduce the size of programs, thereby improving their readability.

File Predicates

Nearly every computer program takes data as input and produces data as output. Programs written in LogiQL are no exception. However, thus far, the only means we have seen for input are delta predicates, and the only ways to produce output are with queries or `lb print` commands.

With real-world programs and their demands for processing *big data*, these devices are clearly inconvenient. This subunit presents a more robust means for dealing with input–output called **file predicates**.

From the point of view of a program, a file predicate is like any other. It contains a sequence of facts. If the file predicate is used for input, then the facts represent exactly the data contained in some existing external file. Conversely, if the predicate corresponds to an output file, then any facts computed by the program and saved into the predicate are also written into the output file.

As a simple example, imagine that we wished to take computed data describing the British monarchs and place it into a *csv* file. *Csv* stands for *comma separated values,* and files in *csv* format are commonly used as a way to import data into spreadsheets.

Table 4.24, copied from the table of birth and death dates given in the consolidation exercise of Chapter 1, is the data we wish to save in the *csv* file, which indicates the dates of birth and death of each of the British monarchs.

The schema for this data is the following:

```
// Schema
Monarch(m), hasMonarchName(m:n) -> string(n).
birthdateOf[m] = d -> Monarch(m), datetime(d).
deathdateOf[m] = d -> Monarch(m), datetime(d).
Monarch(m) -> birthdateOf[m] = _.
birthdateOf[m] = d1, deathdateOf[m] = d2 -> d1 <= d2.
```

TABLE 4.24 Birth and Death Dates for British Monarchs

Monarch	Born	Died
Anne	February 6, 1665	August 1, 1714
George I	May 28, 1660	June 11, 1727
George II	October 30, 1683	October 25, 1760
George III	June 4, 1738	January 29, 1820
George IV	August 12, 1762	June 26, 1830
William IV	August 1, 1765	June 20, 1837
Victoria	May 24, 1819	January 22, 1901
Edward VII	November 9, 1841	May 6, 1910
George V	June 3, 1865	January 20, 1936
Edward VIII	June 23, 1894	May 28, 1972
George VI	December 14, 1895	February 6, 1952
Elizabeth II	April 21, 1926	—

Here are the LogiQL fact assertions for this data:

```
// Data
lang:compiler:disableWarning:DATETIME_TIMEZONE[] = true.
+birthdateOf["Anne"] = #02/06/1665#,
    +deathdateOf["Anne"] = #08/01/1714#.
+birthdateOf["George I"] = #05/28/1660#,
    +deathdateOf["George I"] = #06/11/1727#.
+birthdateOf["George II"] = #10/30/1683#,
    +deathdateOf["George II"] = #10/25/1760#.
+birthdateOf["George III"] = #06/03/1738#,
    +deathdateOf["George III"] = #01/29/1820#.
+birthdateOf["George IV"] = #08/12/1762#,
    +deathdateOf["George IV"] = #06/26/1830#.
+birthdateOf["William IV"] = #08/01/1765#,
    +deathdateOf["William IV"] = #06/20/1837#.
+birthdateOf["Victoria"] = #05/24/1819#,
    +deathdateOf["Victoria"] = #01/22/1901#.
+birthdateOf["Edward VII"] = #11/09/1841#,
    +deathdateOf["Edward VII"] = #05/06/1910#.
+birthdateOf["George V"] = #06/03/1865#,
    +deathdateOf["George V"] = #01/20/1936#.
+birthdateOf["Edward VIII"] = #06/23/1894#,
    +deathdateOf["Edward VIII"] = #05/28/1972#.
+birthdateOf["George VI"] = #12/14/1895#,
    +deathdateOf["George VI"] = #02/06/1952#.
+birthdateOf["Elizabeth II"] = #04/21/1926#.
```

Our first example use of file predicates is to produce as output a *csv* file containing this data. Like any other predicate, a file predicate has a name, and for this example, we will conventionally use the name `_out` for the file predicate providing data to our output file. Also, like other predicates, each file predicate has a sequence of roles, and in this case, each role corresponds to one of the columns of data in the table. In particular, we have roles for the monarchs' names and their dates of birth and death. Moreover, the roles each have a type corresponding to one of LogiQL's primitive types. Hence, for our example the actual declaration for `_out` is

```
_out(s, d1, d2) -> string(s), datetime(d1),
    datetime(d2).
```

From the point of view of the computer system in which the program is running, a file predicate is a file, and it therefore has certain properties, such as its name and the type of data it contains. These properties are expressed in the LogiQL program using functional metapredicates similar to the way we assigned properties (like `Derived`) to predicates. For our example, this code has the following form:

```
lang:physical:filePath[`_out] = "dates.csv".
lang:physical:delimiter[`_out] = ",".
```

The first line in the above code indicates that the name of the file is `dates.csv` and that it resides in the directory in which the program runs. In general, a full path name could be used to designate the file. Note that this filename must be a literal string rather than a computed value. The second line indicates that for this example a comma will be used as the delimiter.

What remains to specify in our program for producing the *csv* file is how the file predicate is actually populated. This can be simply expressed with the following rule:

```
_out(n, d1, d2) <-
    Monarch(m), hasMonarchName(m:n),
    birthdateOf[m] = d1, deathdateOf[m] = d2.
```

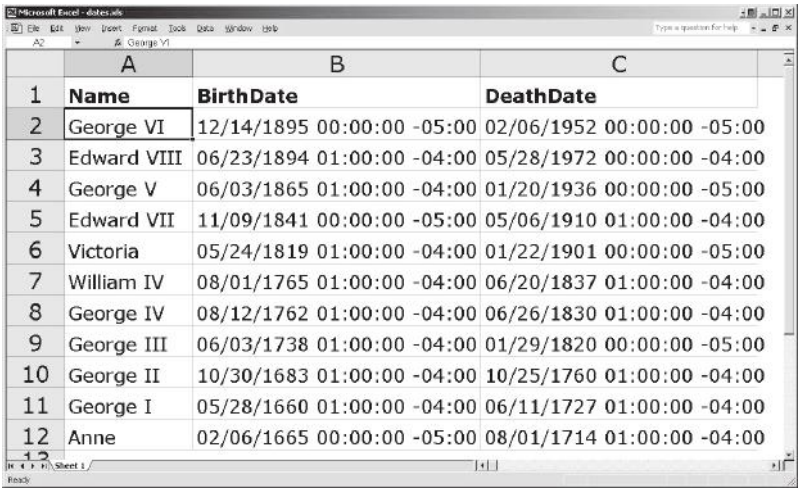
That is, the three values comprising each line in the output are the monarch's name (`n`), the monarch's date of birth (`d1`), and the monarch's date of death (`d2`).

The collected program fragments can be found in the file `datesOutput.lb`, and the resulting *csv* file found in `dates.csv`. The screenshot in Figure 4.7 shows the spreadsheet after the `dates.csv` file contents are imported.

You should note two things about the resulting file. First, not only do the dates of birth and death appear, but also values for the times appear. Of course, we did not provide those in our fact assertions, so the LogiQL engine supplied defaults. The second thing to note is that Queen Elizabeth does not appear in the table. Can you think why she is not there?

She is not there because she is still alive. In particular, the rule computing `_out` requires a death date, and none exists for her.

As another illustration of the use of file predicates, let's now read in data from a file that contains the names of monarchs and their ages at



	A	B	C
1	**Name**	**BirthDate**	**DeathDate**
2	George VI	12/14/1895 00:00:00 -05:00	02/06/1952 00:00:00 -05:00
3	Edward VIII	06/23/1894 01:00:00 -04:00	05/28/1972 00:00:00 -04:00
4	George V	06/03/1865 01:00:00 -04:00	01/20/1936 00:00:00 -05:00
5	Edward VII	11/09/1841 00:00:00 -05:00	05/06/1910 01:00:00 -04:00
6	Victoria	05/24/1819 01:00:00 -04:00	01/22/1901 00:00:00 -05:00
7	William IV	08/01/1765 01:00:00 -04:00	06/20/1837 01:00:00 -04:00
8	George IV	08/12/1762 01:00:00 -04:00	06/26/1830 01:00:00 -04:00
9	George III	06/03/1738 01:00:00 -04:00	01/29/1820 00:00:00 -05:00
10	George II	10/30/1683 01:00:00 -04:00	10/25/1760 01:00:00 -04:00
11	George I	05/28/1660 01:00:00 -04:00	06/11/1727 01:00:00 -04:00
12	Anne	02/06/1665 00:00:00 -05:00	08/01/1714 01:00:00 -04:00

FIGURE 4.7 Spreadsheet with imported monarchy data.

death. In this example, rather than using a *csv* file, we will illustrate how to read from a text file. The file is named `monarchDeathAges.txt`, and it contains the following contents:

```
George VI, 56
Edward VIII, 77
George V, 70
Edward VII, 68
Victoria, 81
William IV, 71
George IV, 67
George III, 81
George II, 76
George I, 67
Anne, 49
```

First, here is the schema that we will use. In particular, we include one property predicate to record the monarchs' ages at death (`deathAgeOf`):

```
// Schema
Monarch(m), hasMonarchName(m:n) -> string(n).
deathAgeOf[m] = a -> Monarch(m), int(a).
Monarch(m) -> deathAgeOf[m] = _.
```

We can describe the input file with the following code:

```
_in[] = s -> string(s).
lang:physical:filePath[`_in] = "monarchDeathAges.txt".
lang:physical:delimiter[`_in] = "\n".
```

The name of the file providing the data is `monarchDeathAges.txt`, and the file predicate that contains its contents is named `_in`. Instead of using a comma to separate the data values into fields, we treat each line as a single fact, with lines delimited by newline characters (`'\n'`). This means that there is no automatic processing of the data, such as looking for commas, as was the case with the *csv* file we populated above. We do the work ourselves to illustrate what you may need to do if an input file's data format does not make use of a delimiter character.

The actual line to read the file contents is similar to what we saw with the *csv* file except that there is only one field, a `string` predicate to hold the contents of each line:

```
+inputString(s) <- _in(s).
```

Once the line is read in, we need to split it into two pieces, and we must make sure that the pieces remain related to each other in the workspace. That is, we want to make sure that Anne's age at death does not somehow become associated with Victoria. To do the former, we make use of the `string:split` function, and the latter can be accomplished using the `seq<< >>` aggregation function. We saw both of these functions in Unit 3.6:

```
line[i] = s -> int(i), string(s).
line[i] = s <- seq<<i = s>> inputString(s).
// The name of each of monarch as a string.
nameString[i] = s -> int(i), string(s).
nameString[i] = n <- string:split[line[i], ",", 0] = n.
// The age of each monarch as a string.
ageString[i] = s -> int(i), string(s).
ageString[i] = d <- string:split[line[i], ",", 1] = d.
// The age of each monarch as an integer.
ageInt[i] = d -> int(i), int(d).
ageInt[i] = d <- ageString[i] = s,
    string:int:convert[s] = d.
```

Here are the rules that actually do the parsing:

```
line[i] = s <- seq<<i = s>> inputString(s).
nameString[i] = n <- string:split[line[i], ",", 0] = n.
ageString[i] = d <- string:split[line[i], ",", 1] = d.
ageInt[i] = d <- ageString[i] = s,
    string:int:convert[s] = d.
```

The `seq` aggregation function associates an index number with each fact in `inputString`. That number is then used with both the name (`nameString`) and the age (`ageInt`). The next two rules use `string:split`. Recall that the three arguments in `string:split`'s keyspace comprise the string to be parsed (`line[i]`), a delimiter character (','), and an integer literal to identify the parsed pieces. The computed result of `string:split` is the text segment identified by the literal. (Recall the convention that the indices into the results of parsing start counting at zero rather than one.)

The fourth rule in the above code converts `ageString` into an `int`, which we name `ageInt`. It makes use of LogiQL's conversion function `string:int:convert`. In general the names of conversion functions are formed by concatenating the names of the two primitive data types involved (`string` and `int` in this case) and appending `"convert"`. The upshot is that `ageInt` is populated with the monarchs' ages at death.

Finally, we can complete our computation by populating the `Monarch` and `deathAgeOf` predicates from our utility predicates:

```
+Monarch(n), +deathAgeOf[n] = d <- nameString[i] = n,
    ageInt[i] = d.
```

After executing this program, we issue the query:

```
_(n, a) <- hasMonarchName(m:n), deathAgeOf[m] = a.
```

we see the expected results:

```
"Anne" 49
"Edward VII" 68
"Edward VIII" 77
"George I" 67
"George II" 76
```

```
"George III" 81
"George IV" 67
"George V" 71
"George VI" 56
"Victoria" 81
"William IV" 71
```

The program code for this example can be found in the file datesInput.lb.

File predicates are a powerful means for dealing with external data. Note, however, that unless the file data are delimited, you are responsible for processing the data to populate your domain predicates. Moreover, the processing of file predicates does little if any error checking. For a robust program, you need to take care of this yourself, either during parsing or via constraints.

Tip: Use derived-only predicates in situations where a rule might generate an infinite amount of data but is only used in other rules that eliminate this possibility.

Tip: Materialize predicates when previously saved results can be reused in computing new results.

Tip: Use hierarchical syntax in situations where a set of related property predicates are being updated.

Tip: Use file predicates to save computed data to files or to assert a large set of facts.

Exercise 5A: Imagine that product prices are stored in predicates using the following schema and rule:

```
Product(p), hasSKU(p:s) -> int(s).
basePriceOf[p] = d -> Product(p), decimal(d). // USD.
taxRate[] = d -> decimal(d).
totalPriceOf[p] = d -> Product(p), decimal(d).
totalPriceOf[p] = d <-
    Product(p),
    (basePriceOf[p] * (1.0 + taxRate[])) = d.
```

Which of these five predicates should be materialized and which should be derived only?

Exercise 5B: Continuing with the example in **Exercise 5A**, imagine further that you wish to sell products in countries that used different currencies, each of which has an exchange rate against the U.S. dollar:

```
Country(c), hasCountryCode(c:n) -> int(n).
currencyOf[c] = s -> Country(c), string(s).
exchangeRateOf[s] = d -> string(s), decimal(d).
localBasePriceOf_In_[p, c] = f -> Product(p),
    Country(c), float(f).
localBasePriceOf_In_[p, c] = f <-
    Product(p),
    Country(c),
    currencyOf[c] = s,
    (basePriceOf[p] * exchangeRateOf[s]) = f.
```

Which of these five predicates should be materialized and which derived only?

Exercise 5C: This exercise makes use of the following schema about wines, which can be found in the file `wineSchema.logic`:

```
// Schema
Wine(w), hasWineId(w:id) -> int(id).
descriptionOf[w] = s -> Wine(w), string(s).
yearOf[w] = y -> Wine(w), int(y).
tasteOf[w] = t -> Wine(w), Taste(t).
colorOf[w] = c -> Wine(w), string(c).
// Red, White, Rose.
quantityOf[w] = n -> Wine(w), int(n).
priceOf[w] = d -> Wine(w), decimal(d).
exclusiveUseOf[w] = b -> Wine(w), boolean(b).

Taste(t), hasTasteId(t:id) -> int(id).
sugarOf[t] = s -> Taste(t), string(s).
// Dry, OffDry, Sweet.
flavorOf[t] = f -> Taste(t), string(f).
// Strong, Moderate, Delicate.
bodyOf[t] = b -> Taste(t), string(b).
// Full, Medium, Light.
```

```
// Constraints
Wine(w) -> colorOf[w] = "Red" ; colorOf[w] = "White" ;
           colorOf[w] = "Rose".
Taste(t) -> sugarOf[t] = "Dry" ; sugarOf[t] = "OffDry" ;
            sugarOf[t] = "Sweet".
Taste(t) -> flavorOf[t] = "Strong" ;
            flavorOf[t] = "Moderate" ;
            flavorOf[t] = "Delicate".
Taste(t) -> bodyOf[t] = "Full" ; bodyOf[t] = "Medium" ;
            bodyOf[t] = "Light".
```

Use hierarchical syntax to assert facts about the two wines described in the file `wineData.txt`.

Exercise 5D: (i) Write a LogiQL program that makes a copy of an input *csv* file comprising lines each of which contains a single `string` value. Try it on the file `input7di.csv`. **(ii)** Write a LogiQL program that makes a copy of an input file treating it as a text file. Test your program on the file `input7dii.txt`.

UNIT 4.6: CONSOLIDATION EXERCISE 4

This exercise gives you a chance to test how well you have mastered the topics covered in this chapter.

Q1: Table 4.25 is an extract of a report that lists kinds of plants. For ease of reference, plant kinds are primarily identified by a plant kind number. However, plant kinds may also be identified by their scientific names. For some plant kinds this is simply their genus names (e.g., *Agrostis*). Some other plant kinds may be identified by their genus and species (e.g., *Acacia interior*). Still other plant kinds may be identified by their genus, species, and infraspecies, which involve both a rank and an infraname (e.g., *Eucalyptus fibrosa* ssp. *nubila*, where ssp. abbreviates *subspecies*).

TABLE 4.25 Plant Species Identifiers

Number	Genus	Species Name	Infraspecies Rank	Infraspecies Infraname
1	Agrostis			
2	Acacia	interior		
3	Eucalyptus	fibrosa	ssp.	nubila

LogiQL code for this report is shown below. The program and data are accessible in the files `Plants.logic` and `PlantsData.logic`. Extend this with derivation rules to derive a printable form of the full scientific name of each plant kind. The answer can be found in the file `Plants2.logic`:

```
// Schema
PlantKind(p), hasPlantKindNr(p:n) -> int(n).
Genus(g), hasGenusName(g:gn) -> string(gn).
Infraspecies(i) -> .
genusOf[p] = g -> PlantKind(p), Genus(g).
speciesNameOf[p] = s -> PlantKind(p), string(s).
infraspeciesOf[p] = i -> PlantKind(p), Infraspecies(i).
rankOf[i] = r -> Infraspecies(i), string(r).
infranameOf[i] = n -> Infraspecies(i), string(n).
PlantKind(p) -> genusOf[p] = _ .
// plantkinds have a genus.
// Each infraspecies has a rank and infraname.
Infraspecies(i) -> rankOf[i] = _, infranameOf[i] = _.
/* Each combination of rank and infraname
   refers to at most one infraspecies */
rankOf[i1] = r, infranameOf[i1] = n ,
    rankOf[i2] = r, infranameOf[i2] = n -> i1 = i2.
// Each plantkind with an infraspecies also has a
// species.
infraspeciesOf[p] = _ -> speciesNameOf[p] = _.

// Data
+genusOf[1] = "Agrostis".
+genusOf[2] = "Acacia", +speciesNameOf[2] = "interior".
+genusOf[3] = "Eucalyptus",
    +speciesNameOf[3] = "fibrosa",
+Infraspecies(i), +infraspeciesOf[3] = i,
+rankOf[i] = "ssp.", +infranameOf[i] = "nubila".
```

Q2: The report in Table 4.26 shows inflation rates in the United States for the years 1991 through 2011. The inflation rate figures show the percentage inflation relative to the previous year. For example, $100 at the end of 1990 is equivalent in purchasing power to $104.23 at the end of 1991, and is equivalent to $107.388 (= $104.23 times 103.03) at the end of 1992.

TABLE 4.26 Inflation Rates by Year

Year	Inflation Rate
1991	4.23
1992	3.03
1993	2.95
1994	2.61
1995	2.81
1996	2.93
1997	2.34
1998	1.55
1999	2.19
2000	3.38
2001	2.83
2002	1.59
2003	2.27
2004	2.68
2005	3.39
2006	3.23
2007	2.85
2008	3.84
2009	−0.36
2010	1.65
2011	3.10

A schema and the associated data for this table are available in the files `Inflation.logic` and `InflationData.logic`, respectively. Using these files, answer the following questions:

(a) Using imperative pseudocode, specify an algorithm that uses a **for-loop** to compute the cumulative inflation since 1990 at the end of 2011. Express this inflation figure as a ratio relative to 1990. For example, the figures for the cumulative inflation since 1990 at the end of the years 1990, 1991, and 1992 are, respectively, 1, 1.0423, and 1.07388.

(b) Using recursion instead of a **for-loop**, write LogiQL code to compute the cumulative inflation ratio at the end of each of the years in the range 1990 through 2011.

(c) Using your program, write a LogiQL query to determine the salary at the end of 2011 that is equivalent in purchasing power, when adjusted for inflation, to a salary of $100,000 at the end of 1990. The answers can be found in `InflationAnswer.logic`.

TABLE 4.27 Employee Parking Data

Employee	Cars	Parking Bay
101	ABC123,BND007	
102		
103	LNX911	5
104	1ABC123	4

Q3: Table 4.27 is extracted from a company report that records which employees drive which cars to work, as well as the parking bay allocated to them (if any). For this application, cars are identified by their license plate numbers.

The following code and data (accessible in `Car.logic` and `CarData.logic`) are the stub of a program and data to model this report:

```
// Schema
Employee(e), hasEmployeeNr(e:n) -> int(n).
Car(c), hasLicensePlateNr(c:n) -> string(n).
ParkingBay(pb), hasParkingBayNr(pb:n) -> int(n).
drives(e, c) -> Employee(e), Car(c).
parkingBayOf[e] = pb -> Employee(e), ParkingBay(pb).

// Data
+drives(101, "ABC123"), +drives(101, "BND007"),
    +parkingBayOf[101] = 3.
+Employee(102).
+drives(103, "LNX911"), +parkingBayOf[103] = 5.
+drives(104, "ABC123"), +parkingBayOf[104] = 4.
```

Although the program works, it lacks the *uniqueness constraint* that each parking bay is allocated to at most one driver and the *equality constraint* that a parking bay is allocated to an employee if and only if that employee drives a car. **(a)** Add code for these missing constraints. The answer can be found in the file `Car2.logic`. **(b)** Modify the program to replace the equality constraint by a *mandatory role constraint* on a role played by drivers. *Hint*: Introduce a subtype for `Driver`. The answer can be found in the file `Car3.logic`.

Q4: (a) Add code to your answer to **(Q3a)**, to enforce the *frequency constraint* that each employee may drive at most two cars to work. Test your constraint by trying to add a third car for employee 101. **(b)** If the *m:n*

predicate `drives(e,c)` is replaced by two functional predicates, `car1Of[e]=c`, and `car2Of[e]=c`, is the frequency constraint from part **(a)** now automatically enforced? The answers can be found in the file `Car4.logic`.

Q5: The report extract shown in Table 4.28 records the membership of mixed doubles teams in tennis. Teams are numbered sequentially based on the week (from week 1 onward) in which a team is to play. In this sense, sometimes the same pair of people may form more than one team. The following code (accessible in `Team.logic` and `TeamData.logic`) is the stub of a program and data to model this report:

```
// Schema
Person(p), hasPersonName(p:pn) -> string(pn).
Man(p) -> Person(p).
lang:isEntity[`Man] = true.
Woman(p) -> Person(p).
lang:isEntity[`Woman] = true.
Team(t), hasTeamNr(t:n) -> int(n).
manInTeam[t] = m -> Team(t), Man(m).
womanInTeam[t] = w -> Team(t), Woman(w).

// Data
+manInTeam[1] = "Alan", +womanInTeam[1] = "Betty".
+manInTeam[2] = "Alan", +womanInTeam[2] = "Cathy".
+manInTeam[3] = "Bob", +womanInTeam[3] = "Cathy".
+manInTeam[4] = "Alan", +womanInTeam[4] = "Betty".
```

Extend the program with code to enforce the *external frequency constraint* that the same pair of people may play together in at most two teams. Test your code with a counterexample. The answers can be found in the file `Team2.logic` and `Team2CounterExample.logic`.

TABLE 4.28 Tennis Team Pairings

Team Number	Man	Woman
1	Alan	Betty
2	Alan	Cathy
3	Bob	Cathy
4	Alan	Betty
...	...	...

TABLE 4.29 Seminar Facility Needs

Seminar	Special Needs	Room
S1	WiFi	10
S2		15
S3	Public address (PA), WiFi	20
S4	PA	20
S5	WiFi	

TABLE 4.30 Room Facilities

Room	Facilities
5	
10	WiFi
15	
20	Public address (PA), WiFi
...	...

Q6: The report extracts shown in Tables 4.29 and 4.30 are maintained by a company that holds seminars. The first report lists each seminar, the room it uses (if decided), and any special facilities needed for the seminar (e.g., wireless Internet access or a public address system). The second report lists each room and its special facilities, if any.

The following code (accessible in `Seminar.logic` and `Seminar-Data.logic`) is the stub of a program and data to model these reports:

```
// Schema
Seminar(s), hasSeminarCode(s:c) -> string(c).
Facility(f), hasFacilityCode(f:c) -> string(c).
Room(r), hasRoomNr(r:n) -> int(n).
needsFacility(s, f) -> Seminar(s), Facility(f).
roomUsedBy[s] = r -> Seminar(s), Room(r).
providesFacility(r, f) -> Room(r), Facility(f).

// Data
+needsFacility("S1", "WiFi"), +roomUsedBy["S1"] = 10.
+roomUsedBy["S2"] = 15.
+needsFacility("S3", "PA"), +needsFacility("S3", "WiFi"),
    +roomUsedBy["S3"] = 20.
+needsFacility("S4", "PA"), +roomUsedBy["S4"] = 20.
+needsFacility("S5", "WiFi").
+Room(5).
```

```
+providesFacility(10, "WiFi").
+Room(15).
+providesFacility(20, "PA"), +providesFacility(20, "WiFi").
```

Add code to enforce the following constraint: If a room is used by a seminar that needs a facility then that room must provide that facility. Include a counterexample to test your code. The answers can be found in the files `Seminar2.logic` and `Seminar2CounterExample.logic`.

Q7: The report extract shown in Table 4.31 concerns bids made by teams for funded projects. Each bid is for a single project and is by a single team. The decision on which bid succeeds for a given project is made by the selection committee for that project.

The following code (accessible in `Bid.logic` and `BidData.logic`) is the stub of a program and data to model these reports:

```
// Schema
Project(p), hasProjectName(p:n) -> string(n).
SelectionCtee(c), hasSelectionCteeNr(c:n) -> string(n).
Person(p), hasPersonName(p:n) -> string(n).
Bid(b), hasBidNr(b:n) -> string(n).
Team(t), hasTeamName(t:n) -> string(n).
isOnCommittee(p, c) -> Person(p), SelectionCtee(c).
isOnTeam(p, t) -> Person(p), Team(t).
projectOfBid[b] = p -> Bid(b), Project(p).
teamOfBid[b] = t -> Bid(b), Team(t).
selectionCteeForProject[p] = c -> Project(p),
    SelectionCtee(c).
```

TABLE 4.31 Funding Team Data

Project	Selection Committee Number	Selection Committee Members	Bids	Bidding Team Name	Bidding Team Members
Mars1	C1	C. Sagan	B1	Trekkers	C. Kirk
		A. Clarke			M. Spock
		I. Asimov	B2	Oldies	F. Gordon
					H. Seldon
					M. Yoda
Jupiter2	C2	A. Clarke	B3	Trekkers	C. Kirk
		H. Solo			M. Spock
		H. Wells	B4	Jedi	L. Skywalker
					M. Yoda

```
// Data
+selectionCteeForProject["Mars1"] = "C1".
+selectionCteeForProject["Jupiter2"] = "C2".
+isOnCommittee("C. Sagan", "C1"),
    +isOnCommittee("A. Clarke", "C1"),
    +isOnCommittee("I. Asimov", "C1").
+isOnCommittee("A. Clarke", "C2"),
    +isOnCommittee("H. Solo", "C2"),
    +isOnCommittee("H. Wells", "C2").
+projectOfBid["B1"] = "Mars1",
    +projectOfBid["B2"] = "Mars1".
+projectOfBid["B3"] = "Jupiter2",
    +projectOfBid["B4"] = "Jupiter2".
+teamOfBid["B1"] = "Trekkers",
    +teamOfBid["B2"] = "Oldies".
+teamOfBid["B3"] = "Trekkers", +teamOfBid["B4"] = "Jedi".
+isOnTeam("C. Kirk", "Trekkers"),
    +isOnTeam("M. Spock", "Trekkers").
+isOnTeam("F. Gordon", "Oldies"), +isOnTeam("H. Seldon",
    "Oldies").
+isOnTeam("M. Yoda", "Oldies"),
    +isOnTeam("L. Skywalker", "Jedi").
+isOnTeam("M. Yoda", "Jedi").
```

Extend the program with code to enforce the *external uniqueness constraint* that for any given project a person may be a member of only one bidding team. Also add code to enforce a *join exclusion constraint* to ensure that a person on a bidding team for a project cannot be on the selection committee for that project. Test your code with counterexamples. The answers can be found in the files `Bid2.logic`, `Bid2CounterExample1.logic`, and `Bid2CounterExample2.logic`.

Q8: The game of tic-tac-toe involves a grid of nine cells formed from three rows and three columns. One player may enter an "O" mark in a cell, and the other player may enter an "X" mark in a cell. Players take turns entering their marks. If a player lines up three of his/her entries in a line (horizontal, vertical, or diagonal) that player wins the game. The diagram in Figure 4.8 shows the grid of a game won by the "X" player. The rows and columns are numbered to allow easy reference. For example, the cell on row 2 and column 3 has the entry "O."

	1	2	3
1	O		X
2		O	O
3	X	X	X

FIGURE 4.8 Tic-tac-toe game board.

The following code (available in `TicTacToe.logic` and `TicTacToeData.logic`) shows a basic program to record the state of play of one game, as well as the data for the example shown:

```
// Schema
entryAt[rowNr, colNr] = e -> int(rowNr), int(colNr),
    string(e).
entryAt[_, _] = e -> e = "0" ; e = "X".

// Data
+entryAt[1, 1] = "0", +entryAt[1, 3] = "X",
+entryAt[2, 2] = "0", +entryAt[2, 3] = "0",
+entryAt[3, 1] = "X", +entryAt[3, 2] = "X",
+entryAt[3, 3] = "X".
```

To better conceptualize the domain, extend the code file available as `TicTacToe.logic` by deriving the entity type `Cell(c)` whose instances are cells in the grid, using the constructor `cellAt[rowNr,colNr]=c` to derive the cells, and `valueOf[c]=v` to return the value entered at cell `c`. *Hint*: Make use of `int:range()` to populate the cell row numbers and column numbers with data. The program, data, and test query are accessible as `TicTacToe2.logic`.

Q9: This exercise question concerns retail sales data from the U.S. Census Bureau. The data are contained in a file named `retailSales.csv`. This file and the other files used in this exercise are found in the `q11` directory and its subdirectories.

(a) Examine the spreadsheet data and prepare a LogiQL schema to describe it. The schema should include an entity predicate (`RawSalesRecord`) whose instances correspond to the rows of the spreadsheet. Place the schema in the file `Schema.logic` and

create a project description file listing it as an `active` block. Check that your schema is correctly described by compiling the project and creating a workspace for it.

(b) Use a file predicate to read the data into a workspace. Place the code to read the spreadsheet data into a file named `readFile.logic`, and use the `lb` command to load the data from the spreadsheet into the workspace.

(c) Note that most of the rows in the spreadsheet contain North American Industry Classification System (NAICS) code numbers. Note also that these code numbers suggest hierarchical groupings. That is, the code 441 describes a class of businesses that is further broken down into subcategories (4411, 4412, and 4413). Define a binary predicate (`childOf`) relating the `RawSalesRecords` of parent categories to those of their children. Use this predicate to define a ternary functional predicate (`totalChildSalesOf`) computing for each parent `RawSalesRecord` and for each year the total sales of all of that business category's children. Define a complementary functional ternary predicate (`parentSalesOf`) that for each category that has subcategories and for each year records that parent's sales. Note that within some groups, such as 441, the sales associated with the subcategories do not add up the sales associated with the category itself. To capture these situations, define a unary predicate (`incompleteParent`) containing `RawSalesRecords` where the total sales of those business categories' children do not completely add up to that of the category itself. Place all of these predicates in a separate logic file named `Children.logic` and include that file in the project description file.

(d) To account for the missing sales, some new sales records must be created. Use a constructor to add a new subcategory code whose last character is an asterisk (e.g., "4481*"), whose "Kind of business" field is "`Other`" and whose sales fields equals the residual amounts not accounted for by the subcategories explicitly listed. Note that your constructor will not work with the `RawSalesRecord` entity because it has a refmode. You can deal with this situation by declaring a new entity type, `ConstructedSalesRecord`, and a constructor for it. You can then combine the two entities (`RawSalesRecord` and `ConstructedSalesRecord`) into a

composite entity (`SalesRecord`) containing both types of sales records, leaving out the refmode values, which are no longer needed. *Hint:* You will need two constructors for `SalesRecord`, one that is keyed by `RawSalesRecord`'s refmode and one that is keyed by `ConstructedSalesRecord`'s NAICS code. You will also need to define auxiliary predicates to hold the sales data and descriptions. Include the declarations for these predicates in a file named `Constructor.logic` and the delta logic that adds the actual data in a separate file named `addRecords.logic`. Be sure to add the former to your project description file.

(e) The retail sales data contained in `retailSales.csv` have not been adjusted for inflation. Using the hierarchical syntax, create a new `SalesRecord` whose NAICS Code and description predicates contain the value "`inflation`" and with inflation values associated with each of the years. Inflation data can be found in the file `inflation.txt`.

Place this code into a separate file named `inflation.logic`, which, because it is delta logic, should not be included in the project description file.

(f) Use the inflation data to adjust the retail sales data. That is, define a ternary functional predicate (`correctedSalesFor`) that for each `SalesRecord` and for each year computes the sales for that year as adjusted for inflation. Because this predicate will be used only in part **(g)**, you should indicate that its values should not be saved (i.e., it should be a `Derived` predicate). Save your code in a file named `Adjust.logic`, and include a reference to the new file in your project description file.

(g) Using the adjusted sales data, compute the compound annual growth rate (CAGR) for each business category. (The CAGR can be computed by the following formula: $CAGR = (V_f/V_i)^{1/t} - 1$ where V_f is the final value, V_i is the initial value, and t is the number of time periods.) That is, define a functional binary predicate (`CAGR`) that, for each `SalesRecord` computes the CAGR for that record, where the CAGR is based on inflation-adjusted sales figures. Save your results in a file named `CAGR.logic`, and add it to your project description file.

(h) Using the growth rates computed in part **(g)**, apply the `max` aggregation function to determine the largest growth rate and the corresponding business category. Save the resulting code in a file named `MaxCAGR.logic` and add its name to your project description file. Note that for the purposes of this exercise, you can assume that there is no more than one store with the maximum growth rate.

(i) Prepare a report using the predicates you have defined. Your report should contain one record with three values: the NAICS code for the category with the greatest growth, its "Kind of business" description, and its growth rate. Save your query in a file named `query.logic`, and add its name to the project description file. *Hint:* Remember to label the block as `inactive`.

(j) Now that you have a project working with `legacy.logic` files, convert it to use modules and concrete blocks. In particular, create a module named `retail`, move the active blocks and query into it, converting them into concrete blocks, and update the project file accordingly. Make sure that the resulting application still runs as you expect.

Solutions containing the described files for all of the parts of this exercise are available in directory `q11` and its `retail` subdirectory.

Q10: In the first exercise of Unit 4.4 you were asked to informally arrange the seating at Snoopy's dinner party. This question asks you to write a LogiQL program for the same task. In particular, for the data given above, your program should produce a report that lists for each table, which three beagles are sitting at that table. That is, the resulting seating arrangement should be such that the total number of friends sitting together is maximized over all possible arrangements. *Hint:* In its general formulation, this problem is called the *uniform graph partition problem (UGPP)*, which can be expensive to compute. However, for nine guests, you should have no problem finding the answer. More information about the UGPP can be found in Wikipedia. *Hint:* You may assume that there is only one best arrangement, which happens to be true for the given data. A solution to this problem can be found in the `q10` directory.

ANSWERS TO EXERCISES

Answer to Exercise 1A:

Additional code:

```
upTo2givenNamesOf[a] = t -> Author(a), string(t).
upTo2givenNamesOf[a] = givenName1Of[a] + " " + gn2 <-
    givenName2Of[a] = gn2.
upTo2givenNamesOf[a] = givenName1Of[a] <-
    !givenName2Of[a] = _.
givenNamesOf[a] = t -> Author(a), string(t).
givenNamesOf[a] = upTo2givenNamesOf[a] + " " + gn3 <-
    givenName3Of[a] = gn3.
givenNamesOf[a] = upTo2givenNamesOf[a] <-
    !givenName3Of[a] = _.
fullNameOf[a] = t -> Author(a), string(t).
fullNameOf[a] = givenNamesOf[a] + " " + familyNameOf[a].
```

The program, data, and test query are accessible as `AuthorNames-Answer.logic`. A query of the results produces the following report:

```
author and full names:
3, Joanne Kathleen Rowling
2, Isaac Asimov
1, John Ronald Reuel Tolkien
```

Answer to Exercise 1B:

Additional code:

```
SmallInteger(n) -> int(n).
SmallInteger(n) <- int:range(0, 20, 1, n).
factorial[n] = x -> int(n), int(x).
factorial[0] = 1.
factorial[n] = n * factorial[n-1] <- n > 0,
   SmallInteger(n).
```

Note the restriction to the specified small integers in the body of the recursive rule. Without this restriction, the rule would be unsafe.

The answer is accessible as `FactorialAnswer.logic`. A query returns the following result:

```
n and n!:
0,  1
1,  1
2,  2
3,  6
4,  24
5,  120
6,  720
7,  5040
8,  40320
9,  362880
10, 3628800
11, 39916800
12, 479001600
13, 6227020800
14, 87178291200
15, 1307674368000
16, 20922789888000
17, 355687428096000
18, 6402373705728000
19, 121645100408832000
20, 2432902008176640000
```

Answer to Exercise 1C:

Additional code:

```
gradeOf[s] = l -> Student(s), string(l).
gradeOf[s] = "A" <- scoreOf[s] = 10.
gradeOf[s] = "B" <- scoreOf[s] > 6, scoreOf[s] < 10.
gradeOf[s] = "C" <- scoreOf[s] = 5 ; scoreOf[s] = 6.
gradeOf[s] = "F" <- scoreOf[s] < 5.
```

The answer is accessible as `GradesAnswer.logic`. A query returns the following result:

```
Students, scores, and grades:
  104, 10, A
  105, 4, F
  106, 7, B
```

```
101, 7, B
102, 9, B
103, 5, C
```

Answer to Exercise 2A:

```
minCelsiusTempOf[c] = _ -> maxCelsiusTempOf[c] = _.
maxCelsiusTempOf[c] = _ -> minCelsiusTempOf[c] = _.
// If city c has a minimum Celsius temperature then it
// has a maximum Celsius temperature, and vice versa.

minCelsiusTempOf[_] = n -> n > -50.
// Each minimum temperature is above -50 degrees
// Celsius.
maxCelsiusTempOf[_] = n -> n > = -10, n < = 50.
// Each maximum temp is in the range -10...50 degrees
// Celsius.
```

The full program is accessible as `CityTempAnswer.lb`.

Answer to Exercise 2B:

```
minCelsiusFor[c, m] = _ -> maxCelsiusFor[c, m] = _.
maxCelsiusFor[c, m] = _ -> minCelsiusFor[c, m] = _.
// If a city in a month has a minimum temperature
// recorded then it has a maximum temperature, and
// vice versa.
```

The full program is accessible as `CityTemp2Answer.lb`.

Answers to Exercise 2C:

(i)

```
positiveNrLanguagesMasteredBy[p] = n <-
    agg<<n = count()>> isFluentIn(p, _).
// If person p is fluent in any language then
// n is the number of languages in which he/she is
// fluent.

isBilingual(p) <- positiveNrLanguagesMasteredBy[p] > 1.
// Each bilingual person has mastered more than
// 1 language.
```

```
!(Translator(p), !isBilingual(p)).
// Nothing can be a translator and not bilingual,
// i.e. each translator is bilingual.
```

Note that it is an error to replace the above constraint by the following code, as the compiler treats that as an attempt to assign two supertypes to `Translator`:

```
Translator(p) -> isBilingual(p). // Error!
```

However, you can avoid this error by adding the following metadeclaration that `isBilingual` is not an entity:

```
lang:isEntity[`isBilingual] = false.
```

The full program is accessible as `LanguagesAnswer.lb`.

(ii)

```
isBilingual(p) <-
    isFluentIn(p, la1), isFluentIn(p, la2),
    la1 ! = la2.
```

For large frequencies, this approach requires far more code than use of the `count` function. For example, to ensure that a person is fluent in at least five languages `la1`, `la2`, `la3`, `la4`, `la5`, we must assert that the person is fluent in each of these languages, and that no two pairs of these languages are equal—that is, `la1 != la2`, `la1 != la3`, `la1 != la4`, `la1 != la5`, `la2 != la3`, `la2 != la4`, `la2 != la5`, `la3 != la4`, `la3 != la5`, `la4 != la5`.

Answer to Exercise 2D:

```
isFluentInAnOfficialLanguageOf(p, c) -> Person(p),
    Country(c).
isFluentInAnOfficialLanguageOf(p, c) <-
    isFluentIn(p, la), hasOfficialLanguage(c, la).
// Person p is fluent in an official language of
// country c if p is fluent in some language la
// that is an official language of c.

countryAmbassadoredBy[p] = c -> isFluentInAnOfficial
    LanguageOf(p, c).
```

```
// If person p is an ambassador to country c
// then p is fluent in some official language of c.
```

Would it be acceptable to replace the above constraint on `countryAmbassadoredBy` by the following constraint?

```
countryAmbassadoredBy[p] = c ->
    isFluentIn(p, la), hasOfficialLanguage(c, la).
```

No. This constraint means something much different. It says that *everything* is an official language of `c` and is mastered by `p`, if `p` is an ambassador to `c`. Moreover, the LogiQL compiler will complain about the construct because it violates the *SC1* safety condition. Can you see why this is so?

SC1 states that all variables in the head of a rule must appear in the body. For constraints, the head is on the right, and `la` does not appear in the body. Another way of looking at this problem is to note that in order for the runtime engine to guarantee that the constraint is met, it has to look at all values of `la`, not just those that correspond to languages. That is, `la` is unbound, thereby violating the intent of *SC1*.

The full program is accessible as `AmbassadorAnswer.lb`.

Answer to Exercise 2E:

```
authoredSomeReviewOf(e, p) -> Employee(e), Product(p).
authoredSomeReviewOf(e, p) <-
    authorOf[r] = e, productReviewedIn[r] = p.
// Employee e authored a review of product r
// if e is the author of r, and the product
// reviewed in r is p.
developed(e, p) -> !authoredSomeReviewOf(e, p).
// If employee e is a developer of product p
// then e did not author a review of p.
```

The full program is accessible as `ProductReviewAnswer.lb`.

Answer to Exercise 3A:

```
Constraint(c), hasConstraintName(c:cn) -> string(cn).
Modality(m), hasModalityName(m:mn) -> string(mn).
hasModalityName( _ :mn) -> mn = "Hard" ; mn = "Soft".
modalityOf[c] = m -> Constraint(c), Modality(m).
```

```
ConstraintVerbalization(cv) ->.
positiveVerbalizationOf[c] = pv ->
    Constraint(c), ConstraintVerbalization(pv).
lang:constructor(`positiveVerbalizationOf).

negativeVerbalizationOf[c] = nv ->
    Constraint(c), ConstraintVerbalization(nv).
lang:constructor(`negativeVerbalizationOf).

modalTextOf[cv] = s -> ConstraintVerbalization(cv),
    string(s).
ConstraintVerbalization(pv),
    positiveVerbalizationOf[c] = pv,
    modalTextOf[pv] = "It is necessary that ",
ConstraintVerbalization(nv),
    negativeVerbalizationOf[c] = nv,
    modalTextOf[nv] = "It is impossible that " <-
    modalityOf[c] = "Hard".
// If a constraint is of hard modality
// then there exists a positive verbalization of c
// whose modal text reads "It is necessary that "
// and there exists a negative verbalization of c
// whose modal text reads "It is impossible that ".

ConstraintVerbalization(pv),
    positiveVerbalizationOf[c] = pv,
    modalTextOf[pv] = "It is obligatory that ",
ConstraintVerbalization(nv),
    negativeVerbalizationOf[c] = nv,
    modalTextOf[nv] = "It is forbidden that " <-
    modalityOf[c] = "Soft".
// If a constraint is of soft modality
// then there exists a positive verbalization of c
// whose modal text reads "It is obligatory that "
// and there exists a negative verbalization of c
// whose modal text reads "It is forbidden that ".
```

The program, data, and test query are accessible as `Verbalize2.lb`.

Answer to Exercise 3B:

```
Playing(p) ->.
playingDerivedFrom[c, s] = p ->
```

```
Country(c), Sport(s), Playing(p).
lang:[`playingDerivedFrom].

sportLinkedTo[p] = s -> Playing(p), Sport(s).
countryLinkedTo[p] = c -> Playing(p), Country(c).
Playing(p), sportLinkedTo[p] = s, countryLinkedTo[p] = c ,
    playingDerivedFrom[c, s] = p <- plays(c, s).
// If country c plays sport s , there exists an object p
// that objectifies this playing and is linked to c
// and s.

rankOf[p] = r -> Playing(p), int(r).
rankOf[p] = r <- playingDerivedFrom[c, s] = p ,
    sportRankFor[c, s] = r.
```

The program, data, and test queries are accessible as `Playing2.lb`.

Answer to Exercise 4A:

```
// Data
+sitsAtTable["Andy"] = 1.
+sitsAtTable["Belle"] = 1.
+sitsAtTable["Missy"] = 1.
+sitsAtTable["Joe Cool"] = 2.
+sitsAtTable["Snoopy"] = 2.
+sitsAtTable["Spike"] = 2.
+sitsAtTable["Marbles"] = 3.
+sitsAtTable["Olaf"] = 3.
+sitsAtTable["World War I Flying Ace"] = 3.
```

Answers to Exercise 4B:

(i) Here is one project description file that works for compiling and installing these files:

```
exercise, projectname
base.logic, active
birthDeath.logic, active
Q1Query.logic, inactive
Q7Query.logic, inactive
Q8Query.logic, inactive
Q9Query.logic, inactive
```

The results can be found in `exercise/i`.

(ii) `Country.logic` looks like the following:

```
block(`Country) {
  export(`{
    Country(c), hasCountryCode(c:cc) -> string(cc).
birthCountryOf[m] = c -> Monarch(m), Country(c).
  }),
clauses(`{
    Monarch(m) -> birthCountryOf[m] = _.
  })
} <- .
```

The project description file should now contain the following line to describe the new module:

```
chapter1CE, module
```

and you need to alter the files `Q1Query.logic` and `birthCountryData.logic` so that references to `birthCountry` now look like `chapter1CE:Country:birthCountry`.

`exercise/ii` is a directory reflecting these changes.

(iii) A solution to this problem involves creating a new concrete block named `Date` contained in the file `Date.logic`. In addition, the project description file can be shrunk by eliminating the files whose contents have been incorporated into `Date.logic`. It now looks like the following:

```
base.logic, active
chapter1CE, module
Q1Query.logic, inactive
Q7Query.logic, inactive
Q8Query.logic, inactive
Q9Query.logic, inactive
```

The resulting files can be found in the directory `exercise/iii`.

(iv) The resulting `Date.logic` file is in `exercise/iv/chapter1CE/Date.logic`.

(v) There are several groups of predicates around which to build concrete blocks. One such group concerns genders and relationships

(`Relations.logic`). Another concerns houses (`House.logic`). The remaining predicates have to do with monarchs and their names (`Monarch.logic`). When you are done constructing these files, the only remaining legacy files should be those concerned with loading data and those containing queries. The modularized project can be found in the directory `exercise/v`.

(vi) Here is a concrete block to replace `Q9Query.logic`.

```
block(`ForeignbornMonarchQuery) {
    inactive(),
    clauses(`{
        q(m) <- chapter1CE:Country:birthCountryO
        f(m, c), c ! = "GB".
    })
} <- .
```

A complete solution for this exercise, converting all of the queries, can be found in the directory `exercise/vi`.

Answer to Exercise 5A:

`Product`, `hasSku`, `basePriceOf`, and `taxRate` should all be materialized. These are predicates holding stored data that will be referred to repeatedly by applications. `totalPriceOf` may be either materialized or derived only. The decision comes down to whether the results will be referred to more than once. If so, it makes sense to materialize the result. If not, then treating this predicate as derived only will save some space and time. The code for this example can be found in `price1.logic`.

Answer to Exercise 5B:

The situation here is similar to that of **Exercise 5A**. `Country`, `hasCountryCode`, `currencyOf`, and `exchangeRate` all concern data that are often used by the application and should therefore be materialized. `localBasePriceOf` is more rarely used. Hence, it is a candidate for being derived only. The code for this example can be found in `price2.logic`.

Answer to Exercise 5C:

```
+Wine(_) {
    +hasWineId(1000),
    +descriptionOf("San Martin Reserve"),
    +yearOf(2007),
    +tasteOf[] = +Taste(_) {
    +hasTasteId(100),
        +sugarOf[] = "Dry",
        +flavorOf[] = "Moderate",
        +bodyOf[] = "Full"
    },
    +colorOf[] = "White",
    +quantityOf[] = 156,
    +priceOf[] = 6.99,
    +exclusiveUseOf[] = false
}.
+Wine(_) {
    +hasWineId(1001),
    +descriptionOf("Saint Ana Chardonnay"),
    +yearOf(2010),
    +tasteOf[] = +Taste(_) {
        +hasTasteId(103),
        +sugarOf[] = "Dry",
        +flavorOf[] = "Strong",
        +bodyOf[] = "Full"
    },
    +colorOf[] = "White",
    +quantityOf[] = 4,
    +priceOf[] = 6.99,
    +exclusiveUseOf[] = false
}.
```

This answer can also be found in the file `wineAnswer.logic`.

Answer to Exercise 5D:

(i) This program can be found in `fileCopy7di.lb`:

```
_in(s) -> string(s).
lang:physical:filePath[`_in] = "input7di.txt".
```

```
_out(s) -> string(s).
lang:physical:filePath[`_out] = "output.txt".

_out(s) <- _in(s).
```

(ii) This program can be found in `fileCopy7dii.lb`:

```
_in[] = s -> string(s).
lang:physical:filePath[`_in] = "input7dii.txt".
lang:physical:delimiter[`_in] = "\n".

_out[] = s -> string(s).
lang:physical:filePath[`_out] = "output.txt".

_out[] = s <- _in[] = s.
```