CHAPTER 1

Basics

CONTENTS

The goal of the first chapter is to quickly get you started writing LogiQL programs. Such a program tries to model aspects of a real-world **domain** inside a computer. The program is written in the LogiQL programming language, a member of the Datalog family of languages, and relevant data are stored in a LogiQL database.

By the end of this chapter, you will have built a working LogiQL program. Your program will include constraints, derivation rules, and fact assertions, and this chapter introduces you to all of these aspects. The program is used to record and derive information of interest about the British monarchy. In writing this program, you should obtain a feel for the structure of LogiQL programs, the basic syntax of the language, and how your program is processed. Later chapters will go into further detail on these topics and introduce you to many other interesting and powerful features of LogiQL.

UNIT 1.1: DATABASES, PREDICATES, AND FACTS

LogiQL is a programming language for accessing logic databases. The data stored in these databases consist of predicates—named collections of related facts. Each fact in a predicate relates the same fixed number of values. For example, the fact `bought("Jim","car")` relates two `string` values, `"Jim"` and `"car"` via the `bought` predicate. The number of values in each fact of a predicate is that predicate's *arity*. For example, `bought` has arity two.

Predicates can be thought of as tables, where each row in the table is a fact, and the number of columns in the table is the predicate's arity. Each column in the table plays a *role* in the predicate. For example, the first column in the `bought` table plays the purchaser role, and the second column plays the `product` role. For each role, all data values in that column have the same datatype.

There are many different kinds of predicates found in LogiQL programs. An *entity* predicate asserts the existence of a set of elements in the problem domain that the program is modeling. The assertion of an entity can be expressed using either of two other kinds of predicates. A *refmode* predicate associates a unique primitive identifying value, such as a string or a number, with each entity. For example, the vehicle identification number of a car serves to uniquely identify it. Alternatively, a *constructor* predicate asserts the existence of an entity as a function of a tuple of values.

Entity predicates often are accompanied by property predicates. A *propery* predicate associates a typed value with each entity. For example, an integer `age` predicate might be associated with a `Person` entity predicate to hold data about people's ages.

The most general class of predicate is the *relation*, which can be used to hold information that associates a typed tuple of values. For example, a relation might assert that particular university courses meet in particular rooms at particular times. A useful kind of relation comprises the functional predicates. A *functional predicate* is a relation in which a subset of the predicate's roles serve as a *key* or index into the predicate's facts. Usually, values of the keys' roles are used to look up the values of the other roles in a predicate's facts. Functional predicates have their own syntax that makes apparent which values serve as the key.

File predicates are one means by which a program can perform input and output (I/O) operations. An input file predicate provides predicate access to the contents of an input file. That is, the facts in the predicate correspond to the contents of the file. Similarly, an output file predicate ensures that the contents of an output file reflect the facts in the predicate.

One final category of predicate contains the *system* or *built-in* predicates. These predicates are provided by the LogiQL runtime engine because they are generally useful, because the engine can compute them more efficiently than can the program, or because they involve LogiQL's primitive datatypes.

Exercise 1: For each of the following descriptions, select the type of predicate most appropriate for expressing it:

1.	the square root function	a.	constructor predicate
2.	data imported from a spreadsheet application	b.	entity predicate
3.	Social Security Numbers of U.S. citizens	c.	file predicate
4.	purchases of cars by individuals	d.	functional predicate
5.	the ages of famous generals when they died	e.	property predicate
6.	descriptions of sports teams in terms of their cities and leagues	f.	refmode predicate
7.	the states of Australia	g.	relation predicate
8.	the year U.S. presidents were elected	h.	system predicate

UNIT 1.2: DECLARING ENTITY TYPES AND REFMODES

Many of the examples in this book are concerned with the British monarchy. In writing programs about a domain like this, you need to make decisions about the data that your program deals with. A key early choice you should think about is which elements of the domain you express as **entities** and which you express as **values**. Roughly speaking, a value is represented in LogiQL using a literal of a built-in datatype, such as a character string (e.g., `"Germany"` or `"Windsor"`). An entity is a concrete object or abstract concept, which you describe with one or more values (e.g., the actual country Germany or the royal house Windsor). Also, most entities can change their state over time. For example, a country may change its average temperature over time, but the string "`Germany`" never changes, even if the country changes its name.

To get started, let's consider how to describe the key entities in the British monarchy domain, the monarchs themselves. In LogiQL, you use an **entity type** declaration to designate a set of similar entities. **Declarations** specify what kinds of entities and facts are of interest and how they are represented in the computer. For example, the Monarch entity type could be declared as follows:

```
Monarch(m) -> .         // Monarch is an entity type.
```

Syntactically, a LogiQL **logic program** comprises a set of clauses, and a declaration is an example of one kind of clause called a *constraint*. A declaration consists of two parts, separated by a right arrow. On the left is a predicate name, giving the name of the entity type, `Monarch`. A predicate denotes a named collection of facts. In this case, the facts indicate the monarchs we refer to in our program.

Predicate names can be followed by a list of arguments, and in this case, there is a single argument, denoted by the identifier `m`, which ranges over individual `Monarch` entities. The combination of a predicate name and its argument terms is called an **atom** (e.g., `Monarch(m)`).

In this example, there is nothing after the right arrow other than a period, which signifies the end of the declaration. That is, there is no information given about how monarch entities are represented in the computer. In fact, the LogiQL engine will handle the internal representation automatically, keeping track of each `Monarch` entity subsequently introduced. The text starting with the two slashes is a comment, which does not have any effect on how the program is evaluated. In the examples, comments are used to suggest how you can verbalize the commented program text.

The above approach for describing monarchs is fairly abstract, and you would normally want to provide some way for people to identify the monarchs rather than relying on artificial internal identifiers created by the system. In database modeling, you would define a *key* in similar situations. In LogiQL, refmodes can be used in circumstances such as this, and a refmode predicate is normally declared at the same time you declare an entity type. For example, here is how you could declare monarchs that are identified by their names:

```
Monarch(m), hasMonarchName(m:s) -> string(s).
/* Monarch is an entity type, and hasMonarchName is a
   refmode predicate for it. A monarch name is
   represented in the computer as a string. */
```

The text on the left of the arrow has several interesting aspects. First, it contains two atoms, separated by a comma (`','`) denoting the logical *and* operator. **Formulas** that connect parts by using *and* operators are called **conjunctions**. Second, the two arguments of `hasMonarchName`, `m` (the monarch entity) and `s` (the monarch name), are separated by a colon. Third, each atom declares a predicate, and the two predicates `hasMonarchName` and `Monarch` share an argument, `m`. Together, these syntactic elements indicate that a special kind of relationship exists between monarchs and their names, in particular that each monarch entity must have exactly one name and that no two monarchs have the same name. Relationships like these not only express actual properties of the domain, but they also give the compiler advice helping it to detect data typing violations and to improve performance.

In the `hasMonarchName` example, there is text to the right of the arrow, `string(s)`. The additional text tells us that `s` is represented in the computer as a string.

Finally, the text following the declaration illustrates a multi-line comment. Multi-line comments start with "/*", span one or more lines, and end with "*/". This kind of comment enables you not only to provide multiple lines of commentary but also to easily comment out a large block of program text.

Note that in this book, we conventionally use uppercase letters to begin the names of entity types and lowercase letters to start the names of non-entity predicates. Also, if the name of any identifier comprises multiple words, we capitalize the first letter of all subsequent words. This convention is called *camelCase*. Be aware that the LogiQL compiler is sensitive to case. That is, `Monarch(m)` is different from `monarch(m)`, and you will see error messages from the compiler if you inadvertently misuse the shift key in typing an identifier's name. These and other conventions are collected in Appendix F.

Tip: When constructing a LogiQL program, for each different kind of entity of interest in the domain, declare an entity type and a refmode.

Tip: Use comments to relate programming elements to the domain elements they represent.

Tip: LogiQL is case sensitive, so be sure to be consistent in your choice of uppercase or lowercase letters when you write a predicate name.

Exercise 2: Members of the British monarchy belong to houses, such as the current house of Windsor. Prepare a LogiQL declaration for the `House` entity type and a corresponding refmode predicate, `hasHouseName`, for the house's name. Do this now before looking at the answer at the end of the chapter.

UNIT 1.3: ENTERING FACTS

Now that we have introduced entity types and refmodes, we can populate a database with actual facts about the British monarchy. For example, we might want to add the fact that George VI was a monarch. We can express this in LogiQL, as follows:

```
+Monarch(m), +hasMonarchName(m:"George VI").
// Add the fact that there is some monarch m,
// where m has the monarch name "George VI"
// (i.e. there is a monarch named "George VI").
```

Note the plus sign ('+') before each predicate name in the above conjunction. This symbol is a **delta modifier** that indicates that the denoted fact should be added to the set of asserted facts. The set of asserted facts for a program is called its **extensional database**, often abbreviated as *EDB*.

There is a shorter way to express the existence of an entity that has a refmode. Instead of the above conjunction, we may simply assert the following:

```
+Monarch("George VI").
// There is a monarch with the monarch name "George VI".
```

The LogiQL compiler reads the above and recognizes that `Monarch's` argument is a literal string rather than a variable. It also knows that `Monarch` has a refmode that is represented by a `string`. It therefore realizes that a shortcut is being taken and substitutes the longer formula given above.

Either of the above approaches asserts facts to the EDB. Those facts are associated with the two predicates `Monarch` and `hasMonarchName`. The former is a unary predicate that is populated by a set of invisible (system-provided) values that denote, within the context of that predicate, the monarchs for whom a fact has been asserted. The latter is a binary predicate relating the specific monarch-name strings with the system-provided values. In both cases, the predicate is said to be an **EDB predicate**.

Tip: Enter facts into the EDB to express what you know about domain entities.

Exercise 3: Prepare LogiQL fact assertions for each of the houses in the British monarchy (Stuart, Hanover, Saxe-Coburg and Gotha, and Windsor).

UNIT 1.4: EXPRESSING DOMAIN RELATIONSHIPS WITH PREDICATES

We have seen in earlier units how to declare entities and to express simple facts about them. Using LogiQL, it is possible to describe more general relationships among the domain elements. For example, in the British monarchy domain, monarchs may be identified by their monarch name (e.g., "Elizabeth II"), but they were also given names at birth, one of which was chosen as the basis of their monarch name. As you can see from Table 1.1, a monarch may have many given names, and the same given name may be used by many monarchs. This association between monarchs and given names is an example of a *many-to-many relationship*. In order to express the facts represented in this table, it's best to first verbalize some of them in natural language. How would you verbalize the information indicated by the predicate fragment shown in Table 1.2?

Because we are familiar with the domain, we can see that the two column entries are related to each other. Assuming that we are not interested in the order of the given names, we might verbalize this connection as follows:

The monarch named "George I" has the given names "George" and "Louis."

Note, however, that the above sentence is really expressing two facts. In general, you should ensure that the facts you assert are atomic, in the sense that they cannot be expressed as conjunctions of smaller facts without losing information. This makes it easier to avoid redundancy and

TABLE 1.1 Given Names of British Monarchs

Monarch	**Given Names**
Anne	Anne
George I	George, Louis
George II	George, Augustus
George III	George, William, Frederick
George IV	George, Augustus, Frederick
William IV	William, Henry
Victoria	Alexandrina, Victoria
Edward VII	Albert, Edward
George V	George, Frederick, Ernest, Albert
Edward VIII	Edward, Albert, Christian, George, Andrew, Patrick, David
George VI	Albert, Frederick, Arthur, George
Elizabeth II	Elizabeth, Alexandra, Mary

TABLE 1.2 Given Names for George I

Monarch	**Given Names**
George I	George, Louis

make changes later on. Because the above statement is compound rather than atomic, it should be rephrased as the following two facts:

The monarch named "George I" has the given name "George."

The monarch named "George I" has the given name "Louis."

We may declare facts of this sort in the following way:

```
hasGivenName(m, gn) -> Monarch(m), string(gn).
// If m has the given name gn
// then m is a Monarch and gn is a string.
```

Note that when you declare a **many-to-many predicate** like `hasGivenName`, its arguments are enclosed in parentheses and separated by a comma, unlike refmode predicate declarations where the arguments are separated by a colon. Informally, we sometimes refer to predicates like `hasGivenName` as **property predicates**, to distinguish them from entity predicates and refmode predicates. Intuitively, property predicates are used to indicate properties of known entities.

Once we have declared the `hasGivenName` predicate, we can express the facts about George I's given name as follows:

```
+hasGivenName("George I", "George"),
    +hasGivenName("George I", "Louis").
// George I has given name George and
// George I has given name Louis.
```

Note that if you assert either of these facts about George I's given names, there is no need to explicitly assert the following fact, because it can be inferred from the given name fact and the type declaration for the `hasGivenName` predicate.

```
+Monarch("George I").
```

Tip: Express many-to-many relationships with many-to-many predicates.

Exercise 4: `hasGivenName` is an example of a many-to-many predicate. This means both that monarchs may have many given names and that a single given name may belong to many monarchs.

Exercise 4A: Have a look at the table of given names at the start of this unit, and determine which monarch has the most given names.

Exercise 4B: Have a look at the table of given names at the beginning of this unit, and determine which given name belongs to the most monarchs.

UNIT 1.5: CONSTRAINING THE DATA

Unit 1.4 discussed how to declare predicates to model facts about the British monarchy. The entity types were modeled as the unary predicates `Monarch(m)` and `House(h)`, and their instances were identified using the refmode predicates `hasMonarchName(m:s)` and `hasHouseName(h:s)`. Relationships between monarchs and their given names were modeled using the many-to-many predicate `hasGivenName(m,gn)`. However, there are restrictions on the above data that have not yet been expressed. For example, each monarch must have at least one given name. If we were entering monarchy data ourselves or importing monarchy data from an external source, we would want to make sure that the imported data does not violate this restriction. In this unit we discuss how to explicitly represent such restrictions using constraints.

The following code shows how to declare the constraint that each monarch has a given name:

```
Monarch(m) -> hasGivenName(m, _).
// If m is a monarch then m has some given name.
```

Syntactically, a constraint looks like a declaration. The left-hand side indicates the predicate being constrained, and the right-hand side indicates the properties that facts about the corresponding entity must obey.

In this constraint, the **anonymous variable**, depicted by an underscore (‘ _ ’), is read as “something.” We use the anonymous variable, because in this example, we do not care what the given name is, only that it must exist.

The above constraint is an example of a simple **mandatory role constraint**, since it declares that the role of having a given name is mandatory for each monarch. We use the term **role** to mean a part played in a relationship. For example, a binary predicate has two roles, one for each argument position.

With this constraint in place, if you try to add a monarch without any given names you will get a constraint violation error, for example,

```
+Monarch("George I").            // Error!
// George I is a monarch, but no given name is declared.
```

Note that, in this example, even though King George I was given a monarch name, there is no explicit given name asserted.

Tip: If you wish to place a specific restriction on how non-refmode predicates may be populated with data, then declare a constraint to express that restriction.

Tip: Refmode predicates are understood to be mandatory for their entity type, so there is no need to separately code a mandatory role constraint for them.

Tip: If you do not care what value a variable holds and it is only used once, use the anonymous variable (' _ ').

Exercise 5A: Assume that a predicate `isOfHouse(m,h)` is used to record membership of monarchs in royal houses. Prepare a LogiQL mandatory role constraint guaranteeing that each monarch is a member of some royal house.

Exercise 5B: Add a constraint to the `isOfHouse(m,h)` predicate to ensure that each monarch belongs to at most one house. *Hint 1:* If two facts both assert that a given monarch belonged to a house, the two houses must be the same. *Hint 2:* To indicate that the values of two variables are the same in LogiQL, the *equals* operator ('=') can be used. *Note:* In the next unit, we discuss an easier way to express this constraint.

UNIT 1.6: DECLARING FUNCTIONAL PREDICATES

To date, there have been 12 British monarchs belonging to four houses. Unit 1.2 discussed how to declare the `Monarch` and `House` entity types and the refmode predicates used to identify their instances. Unit 1.3 showed how to add instance data about these entities. Exercises 5A and 5B showed one way to relate the monarchs to their houses. In this unit we'll see an alternative way to declare a predicate that can express the house membership facts. The predicate is written using LogiQL's **functional notation**.

As you can see from the data in Table 1.3, although many monarchs may belong to the same house, each monarch may belong to only one house. This association between monarchs and their houses is said to be *functional*, because the house can be determined from the monarch.

TABLE 1.3 British Monarchy `houseOf` Predicate

Monarch	**House**
Anne	Stuart
George I	Hanover
George II	Hanover
George III	Hanover
George IV	Hanover
William IV	Hanover
Victoria	Hanover
Edward VII	Saxe-Coburg and Gotha
George V	Windsor
Edward VIII	Windsor
George VI	Windsor
Elizabeth II	Windsor

If a predicate is functional and not a refmode predicate, its arguments should be declared using LogiQL's functional notation, in which the arguments that functionally determine the final argument (the **keyspace**) are placed in square brackets, followed by the *equals* operator and the final argument. Those arguments not in the keyspace of a predicate are said to form its **valuespace**. In the case of the `houseOf` predicate, the first argument, `m`, a `Monarch` entity, is the only member of the keyspace, and the result, `h`, a `House` entity, comprises the valuespace.

For example, the house membership predicate may be declared as follows:

```
houseOf[m] = h -> Monarch(m), House(h).
// If the house of m is h
// then m is a monarch and h is a house.
// Each monarch m is of at most one house.
```

The square brackets in the above declaration indicate the functional nature of the relationship between a monarch and his/her house. The use of the word `Of` in the name of the predicate emphasizes the connection between a monarch and that monarch's house.

Similarly to how we asserted facts about entities in the previous unit, we can assert facts for functional predicates. For example, we can indicate that William IV belonged to the House of Hanover as follows:

```
+houseOf["William IV"] = "Hanover".
// William IV is of the house of Hanover.
```

Although the functional, square-bracket notation should be used in the program code for the functional predicate, if you prefer you may use the semantically equivalent parenthesis notation as an alternative when adding data, for example,

```
+houseOf("William IV", "Hanover").
// William IV is of the house of Hanover.
```

Contrast the functional approach described here to that of Exercises 5A and 5B, in which house membership is declared using a parenthesized argument list as follows:

```
isOfHouse(m, h) -> Monarch(m), House(h).
// If m is of house h
// then m is a monarch and h is a house.
```

Note that although `isOfHouse` and `houseOf` appear to serve the same purpose, there is a subtle difference. Implicit in the monarch domain is the constraint that no monarch can belong to two houses. To express this constraint with `isOfHouse`, we would have to explicitly add the following constraint:

```
isOfHouse(m, h1), isOfHouse(m, h2) -> h1 = h2.
// If m is of house h1 and m is of house h2 then h1
// and h2 are the same.
```

With the functional notation, however, this constraint is implicit. That is, because we used the functional notation, the predicate's functional nature is automatically declared.

Tip: Use functional notation to express functional associations.

Exercise 6: Consider another property of monarchs, their genders. Table 1.4 expresses this information for the British monarchs, using the gender codes "M" (for male) and "F" (for female).

Exercise 6A: Declare an entity predicate (`Gender`) and a refmode predicate for it (`hasGenderCode`) that represent the gender code by a string.

TABLE 1.4 Genders of British Monarchs

Monarch	**Gender**
Anne	Female (F)
George I	Male (M)
George II	M
George III	M
George IV	M
William IV	M
Victoria	F
Edward VII	M
George V	M
Edward VIII	M
George VI	M
Elizabeth II	F

Exercise 6B: Declare a constraint guaranteeing that each monarch is either male or female. *Hint 1:* Recall from Unit 1.5 how to use anonymous variables to indicate that a constraint holds for all monarchs. *Hint 2:* In situations like the above, where there are two possibilities, you can use the *inclusive-or* operator, denoted by a semicolon (‘`;`’). Such situations are called **disjunctions**.

Exercise 6C: Declare a functional predicate `genderOf[m]=g` reporting the gender `g` for monarch `m`.

Exercise 6D: Provide explicit facts asserting that Anne I is a monarch with monarch name “Anne I,” that there is a gender code “F,” and that Anne I has that gender.

Exercise 6E: Use the shortened form of fact entry described in Unit 1.3 to assert that George I is a male monarch.

UNIT 1.7: DECLARING DERIVATION RULES

In a typical application, some facts are simply asserted to be true, using the delta modifiers presented in Unit 1.3, while other facts are computed by applying a derivation rule to facts that are already known. For example, if we assert the length and breadth of a window, we can derive the window’s area by multiplying its length by its breadth.

In the current unit, we discuss how to express some basic derivation rules in LogiQL. As a simple example, recall that earlier we represented gender data for monarchs using the functional, binary predicate `genderOf[m]=g`. Instead, we might have declared unary predicates to express the same information as follows:

```
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.
```

Choosing this approach has a subtle implication. Recall that `genderOf` is a functional predicate. This means that for each value of argument `m`, there can be at most one `g` such that `genderOf[m]=g`. In other words, no monarch can have two genders.

Because `isMale` and `isFemale` are separate, unary predicates, we have to explicitly eliminate the possibility of someone being both male and female at the same time. We can indicate this kind of mutual exclusion by using an **exclusion constraint**. Here is one way to express this:

```
isMale(m) -> !isFemale(m).
// If m is male then m is not female.
```

Note the use of the exclamation mark ('`!`') for the logical *not* operator. Note also that there is no need to add the following constraint, since it is implied by the above constraint:

```
isFemale(m) -> !isMale(m).
// If m is female then m is not male.
```

Because the same information is being expressed by `genderOf` and the new predicates, we should be able to derive `isMale` and `isFemale` facts from `genderOf` data. For example, if we assert the following fact:

```
+genderOf["Edward VII"] = "M".
```

we should be able to derive the fact

```
isMale("Edward VII").
```

To do this we can declare a simple rule that can perform this kind of derivation for all the male gender facts as follows:

```
isMale(m) <- genderOf[m] = "M".
// m is male if m has the gender with gender code "M".
```

Note that the direction of the arrow in rules is opposite to what we have seen before. An arrow directed to the right is used in constraints, whether its purpose is to declare a predicate or to limit the facts that can populate one. To visually distinguish rules, their arrows point leftward. Both arrows indicate a conditional dependency that may be expressed using a phrase containing the words "if" and "then." That is, "if" the formula that comes after the arrow is true, "then" the formula before the arrow will also be true.

The formula on the left-hand side of "<-" is called the **rule head**, and the formula on the right-hand side of "<-" is called the **rule body**. The rule specifies the following: for each value of m where genderOf[m] has the value "M", the following fact is derived: isMale(m).

From a programming point of view, the above rule searches for genderOf facts having the string "M" in the value role. For each such fact found, a corresponding isMale fact is derived. Predicates computed from rules, like isMale, which are explicity asserted are called **IDB predicates** to distinguish them from EDB predicates like genderOf.

A more complex derivation rule can be demonstrated using the parenthood graph shown in Figure 1.1. If a monarch is a parent of another monarch, this is shown as a line connecting the parent to the child below it.

Because this graph is confined to the 12 British monarchs, at most one parent is shown for each monarch. We can assert the eight parenthood facts conveyed by this graph using the functional predicate parentOf, which may be declared as follows:

```
parentOf[m1] = m2 -> Monarch(m1), Monarch(m2).
// If the parent of m1 is m2, then m1 and m2 are
// monarchs.
```

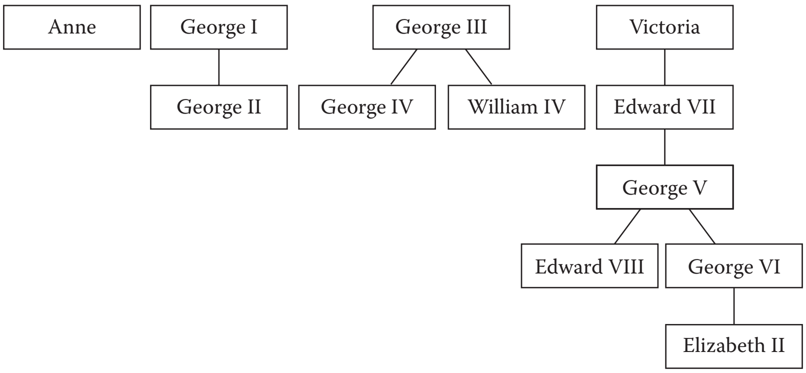

FIGURE 1.1 Parenthood relationship among British monarchs.

Note that in a wider domain where both of a person's parents may be recorded, parenthood would instead be modeled as an *m:n* predicate such as `hasParent(p1,p2)` or `isParentOf(p1,p2)`.

Now consider a rule to derive the sibling relationship. Two different people are siblings of each other if they share a parent. For example, George IV and William IV are siblings. Because it is possible for a monarch to have more than one sibling monarch, we express the siblinghood relationship as an *m:n* predicate rather than as a functional predicate, using the following rule:

```
isSiblingOf(m1, m2) -> Monarch(m1), Monarch(m2).
isSiblingOf(m1, m2) <- parentOf[m1] = m3,
    parentOf[m2] = m3, m1 ! = m2.
// m1 is a sibling of m2 if there is some m3 such that
// m1 and m2 have m3 as a parent, and m1 is not the
// same as m2.
```

Note that in this rule, the rule body introduces a new variable, `m3`, that does not occur in the head of the rule. Occurrences of variables in the rule body that do not appear in the rule head are treated specially. In particular, variables that occur only in the body are assumed to have at least one existing instance. In this example, `m3` is such a variable. It is as if the phrase "there is some `m3` such that" is inserted before the conjunction of three conditions in the rule body.

In the above example, inequality is expressed using the *not-equals* operator ('`!=`'). If we omitted the final check that uses it (`m1 != m2`), then each monarch would be his/her own sibling!

One other thing to observe about the `isSiblingOf` example is that it does not include atoms, such as `Monarch(m1)`, limiting `isSibling Of's` arguments. In this case, the LogiQL compiler is able to infer the types of `m1` and `m2` because of their use in the `parentOf` atoms in the rule's body. The compiler is able to make such inferences for derivation rules in most cases. However, if you see an unexpected error message from the compiler, it is always okay to include an explicit mention of the type for the arguments of the predicate in the derivation rule.

One final example shows how a derivation rule may use an IDB predicate in its body. The following rule invokes two derived predicates to derive a third:

```
isBrotherOf(m1, m2) <- isSiblingOf(m1, m2), isMale(m1).
// m1 is a brother of m2 if m1 is a sibling of m2 and
// m1 is male.
```

Recall that for any given application domain, the set of facts that are simply asserted is known as the extensional database (EDB). In contrast, the set of facts that are inferred from other facts via derivation rules is called the intensional database (IDB). Whereas asserted facts must be explicitly managed (i.e., retracted, updated, etc.) by the programmer, derivation rules are automatically invoked whenever a workspace change affects predicates in the rules' bodies.

Tip: Use derivation rules to express computed domain relationships.

Exercise 7A: Compose LogiQL fact assertions to express the parenthood data illustrated in Figure 1.1.

Exercise 7B: Write a derivation rule to derive `isFemale` from `genderOf`.

Exercise 7C: Write a derivation rule to derive `fatherOf[m1]=m2`.

Exercise 7D: Write a derivation rule to derive `isSisterOf(m1,m2)`.

Exercise 7E: Write a derivation rule for a predicate named `hasNoMonarchSibling` to derive those monarchs who have no monarchs as siblings. Use the parenthood data you entered for **Exercise 6A** to determine which monarchs satisfy this rule.

Exercise 7F: For which pairs of monarchs is the predicate `isBrotherOf` true?

UNIT 1.8: QUERYING A WORKSPACE

Once you have written your program, loaded it into a workspace, and entered relevant facts, you will want to see the results. The `lb` command, described more completely in Appendix A, provides several ways for accessing this information. For example, the following `lb` command prints out the contents of the *predicateName* predicate:

```
lb print workspaceName predicateName
# Print out the current contents of the predicate
# named in the workspace named.
```

Sometimes, we may want to find out about information in the workspace that relates to more than one predicate. You can do this by issuing a **query**.

A query to `lb` makes use of the `exec` option, but instead of asking `lb` to execute the contents of a file, you can supply a LogiQL rule. In this case, the rule you provide indicates that a new predicate, occurring on the left-hand side of the arrow, should be populated with all facts satisfying the right-hand side of the rule.

Conventionally, the predicate on the left-hand side is an **anonymous predicate**, designated with an underscore (' _ '), optionally followed by other characters allowed in an identifier. Note that this use of underscore indicates an anonymous predicate, where earlier we used it to indicate an anonymous variable. Because predicates are followed by arguments, you can always tell which use of underscore is intended.

Here is how `lb` can be used to express a query:

```
lb exec workspaceName '_(args) <-
    predicateName(args, "someString").'
# Using the workspace named workspaceName, locate
# those facts in the predicate named predicateName
# whose final role is filled with the literal value
# "someString" and print out the corresponding
# values that fill the other roles.
```

Tip: Use the '`.logic`' filename extension to name your LogiQL program and fact files.

Tip: Use an anonymous predicate to construct workspace queries.

Exercise 8A: Create a new workspace named `ws`. *Hint:* Refer to Appendix A if needed.

Exercise 8B: Add the rules in the file `base.logic` to this workspace. The file contains the rules we have seen thus far in the chapter.

Exercise 8C: Add the facts in the `baseData.logic` to this workspace. The file contains the facts we have seen thus far in the chapter.

Exercise 8D: Use `lb` to print out the names of the houses in the British monarchy.

Exercise 8E: Execute a query on this workspace to list each female monarch and her royal house.

Exercise 8F: Execute a query on this workspace to list each monarch who has either "George" and/or "William," but not "Albert" as a given name.

Exercise 8G: Execute a query on this workspace to list each monarch who has at least three given names.

UNIT 1.9: CONSOLIDATION EXERCISE 1

Thus far, you have been introduced to LogiQL via a series of small examples—individual declarations, facts, constraints, rules, and queries—but applications are not only larger than this, their pieces are more interdependent. This unit asks you to integrate what you have learned so far to produce a comprehensive program for providing information about the British monarchy.

The exercise begins with what you have done already. The file `base.logic` contains the declarations, constraints, and rules that were introduced during the course of Chapter 1; the file `baseData.logic` contains the facts. In this exercise you will add to these files using a text editor and test them using `lb`, as described in Unit 1.8 and Appendix A. At this point, please create a new workspace called `ws`, load in `base.logic`, and execute `baseData.logic`.

During the course of this exercise, you will be asked to prepare new declarations, rules, constraints, and facts. You should place these into appropriately named files using a text editor and then use `lb` to include them in the workspace you have created. You should use `lb addblock` for new rules and constraints, and for new facts you should use `lb exec`. Depending on the specific task you are asked to perform, you may need to recreate your workspace and reload your code. You should be particularly careful to do this if your previous test resulted in an error. Another situation to be aware of is when you add a constraint to the program installed in your workspace that requires certain facts to pertain, but you have not yet asserted those facts.

PART 1: COUNTRY OF BIRTH

The first extension that we would like to consider is information about the countries in which the British monarchs were born: It turns out that two of them were actually born in Germany! The relevant data is shown in Table 1.5.

Q1a: Extend the program in `base.logic` by declaring the entity predicate `Country(c)`, the refmode predicate `hasCountryCode(c:cc)`, and the property predicate `birthCountryOf[m]=c` to enable facts to be stored about monarch birth countries. Also include a constraint to

TABLE 1.5 Birth Countries of British Monarchs

Monarch	**Birth Country**
Anne	Great Britain (GB)
George I	Germany (DE)
George II	DE
George III	GB
George IV	GB
William IV	GB
Victoria	GB
Edward VII	GB
George V	GB
Edward VIII	GB
George VI	GB
Elizabeth II	GB

ensure that each monarch was born in some country. The answer can be found in the file `Q1Answera.logic`.

Q1b: Use `lb` to add the information in the above table as new facts to `baseData.logic`. The answer can be found in file `Q1Answerb.logic`.

Q1c: You should now be able to write a query to determine which monarchs were not born in Britain. The answers can be found in file `Q1Answerc.logic`.

PART 2: BIRTH AND DEATH DATES

More interesting and more ambitious than incorporating monarchs' birth countries is keeping track of important dates for them, such as their birth and death dates. Table 1.6 (available in `birthDeathData.logic`) provides this data for the British monarchs.

Using the techniques that we have already seen, we could encode these dates as `strings`. This approach would prove difficult, however, once we started doing computations on the dates, such as determining how old the monarchs were when they died. Fortunately LogiQL has a way around this difficulty using the `datetime` primitive datatype.

Using `datetime`, we can declare a predicate expressing the information in the second column of the table as follows:

```
birthdateOf[m] = d -> Monarch(m), datetime(d).
// If m was born on d then m is a monarch
// and d is a datetime value.
```

TABLE 1.6 Birth and Death Dates of 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 is a corresponding constraint that guarantees that every monarch has a birthday:

```
Monarch(m) -> birthdateOf[m] = _.
```

Additionally, we can assert Anne's date of birth as follows:

```
+birthdateOf["Anne"] = #02/06/1665#.
// Anne was born on 6 February, 1665.
```

Note that literal `datetime` values are surrounded by hash symbols ('#'). In particular, in this example, dates are expressed using the format `#mm/dd/yyyy#`, where *mm* is the month number, *dd* is the day number, and *yyyy* is the year number. Be aware that the order of these three values is different from the order in which they were presented in the above table.

If you run the above code, you may see a warning message from the LogiQL compiler indicating that timezone information is missing from the `datetime` literals. If you wish to suppress the display of this warning, include the following line of code in your source file:

```
lang:compiler:disableWarning:DATETIME_
  TIMEZONE[] = true.
```

Q2a: Declare a predicate `deathdateOf` indicating the `datetime` of a monarch's death, and assert Anne's date of death. The answer can be found in the file `Q2Answera.logic`.

Q2b: Using the *less than or equal to comparison* operator ('<='), add a constraint to require that a monarch's death date must come no earlier than the monarch's birth date. (Note that from here on out, we will not be specifically reminding you to run `lb`, but good coding practice says you should always test your results.) The answers can be found in the file `Q2Answerb.logic`.

Note that in **Q2**, you were not asked to provide a constraint guaranteeing that each monarch has a date of death. Can you see why doing so would be a problem?

Of course, it's because Queen Elizabeth II is very much alive!

Given this difference between the `birthdateOf` and `deathdateOf` predicates, we might like to know, for a given monarch, if that monarch is dead.

Q3: Write a derivation rule, `isDeadMonarch`, to determine whether a monarch is dead. The answer can be found in `Q3Answer.logic`.

If we can determine if a monarch is dead using `isDeadMonarch`, it is natural to ask the inverse question—is a monarch alive? We can do this as follows:

```
isLiveMonarch(m) <- Monarch(m), !isDeadMonarch(m).
// m is a live monarch if m is a monarch and m is not
// dead.
```

Would it be acceptable to shorten this rule as follows?

```
isLiveMonarch(m) <- !isDeadMonarch(m).
```

No! This shorter rule implies that anything that does not have a death date is a live monarch. For example, the house of Tudor would be a live monarch. So you need to be careful when using negation to properly constrain the objects under discussion to the entity types that you intend.

PART 3: AGE AT DEATH

As a more complex derivation example, we may compute the age at death for monarchs from their dates of birth and death. The derivation rule for death ages is complicated by the need to consider not just the year, but the month and day values. Think for a moment how you would go about expressing this rule.

As you probably determined, you can derive the death age by subtracting the birth date from the death date, extracting the number of years, and then compensating for monarchs who died during a year before having had their birthday that year.

Table 1.7 illustrates the possibilities.

Note that Anne was born in February and died in a later month (August). Her age at death can be easily computed by subtracting her birth year, 1665, from her death year, 1714, resulting in an age at death of 49. Edward VII, however, died in May, well before his birth month of November. If we tried to simply subtract 1841 from 1910, we would erroneously compute 69. Instead we have to recognize this situation and compensate by subtracting an additional year.

The problem is even worse than indicated so far. Consider the data for George II shown in Table 1.8.

Note that he was born and died in the same month, October. Hence, we have to look to see when in the month these two events took place. Of course, we could go further and look at the time of day or even the time zone in which the monarchs were born and died. But, for this exercise, we will be satisfied with the above degree of precision.

To perform the age-at-death computation, we have to be able to extract the month and day information from `datetime` values. Fortunately, LogiQL has built-in functions for working with `datetime` data. For example, the `datetime:part` function can be used to extract the day, month, and year parts of a date, and the `datetime:offset` function may be used to return the difference between two dates in a specified duration unit. By the way, the names of these two functions share a common prefix, `"datetime:"`, suggesting that the two functions are members of a group of related functions. Note also that this use of the colon is distinct from its use declaring refmode predicates that we saw in Unit 1.2.

TABLE 1.7 Ages at Death of British Monarchs

Monarch	**Born**	**Died**	**Age at Death**
Anne	February 6, 1665	August 1, 1714	49
Edward VII	November 9, 1841	May 6, 1910	68

TABLE 1.8 Age at Death of George II

Monarch	**Born**	**Died**	**Age at Death**
George II	October 30, 1683	October 25, 1760	76

While the whole process of computing the age at death could be formulated as a single derivation rule, the solution is easier to construct and understand if we break the problem into smaller steps, using intermediate predicates to help with deriving later ones. Hence, we begin by extracting the month-of-the-year information from a `datetime` value. In particular, here is a derivation rule for determining the number (from 1 to 12) of the month in which a person was born:

```
birthMonthNrOf[m] = n <-
    birthdateOf[m] = d,
    datetime:part[d, "month"] = n.
// The birth month number of monarch m is n if m was
// born on the date d, and the month part of d is n.
```

The `datetime:part` function has two arguments. The first is a `datetime` value and the second is a `string`, in this case "`month`," specifying which part of the `datetime` value is to be returned.

Q4: Formulate derivation rules for `deathMonthNrOf`, `birthDayNrOf`, and `deathDayNrOf`. *Hint:* For the last two, you should use `"day"` as the second argument to `datetime:part`. The answers can be found in `Q4Answer.logic`.

Our next step is to determine whether a person's calendar day of death occurs before his/her calendar day of birth. This is true if one of the following conditions holds: (a) the death month number precedes the birth month number (like George IV); (b) the death and birth month numbers match, but the death day number precedes the birth day number (like George II).

We can express these conditions using our newly defined rules as follows:

```
hasDeathdayBeforeBirthday(m) <-
    (deathMonthNrOf[m] < birthMonthNrOf[m]) ;
    ((deathMonthNrOf[m] = birthMonthNrOf[m]),
    (deathDayNrOf[m] < birthDayNrOf[m])).
// Monarch m has deathday before birthday if either
// m's deathmonth number is less than m's birthmonth
// number or the two numbers are equal and m's
// deathday number is less than m's birthday number.
```

Note the parentheses on the right-hand side of the above rule. They are used to ensure that the two requirements of the second alternative are both met.

We can now make use of the `datetime:offset` function to estimate the age at death. This function takes three arguments. The first two are `datetime` values, and the third is a `string` indicating the unit of duration we are interested in. In our case, this is `"years"`. The result is an estimate, however, because we may have to adjust the value if indicated by `hasDeathdayBeforeBirthday`.

```
approxDeathAgeOf[m] = n <-
    birthdateOf[m] = d1, deathdateOf[m] = d2,
    datetime:offset[d1, d2, "years"] = n.
// Monarch m's approximate death age is n if m was born
// on datetime d1 and died on datetime d2, and the
// offset between these two values was n years.
```

We can express the derivation rule that computes the age at death using `approxDeathAgeOf` and adjusting the computed value, if necessary, by `hasDeathdayBeforeBirthday`. Specifically, if a person's calendar day of death is before his/her calendar day of birth, then the death age is the approximate age minus one year. Otherwise, the death age is the approximate death age. This rule has the form *if p then q else r*, where *p*, *q*, and *r* are assertions. In LogiQL, such rules are expressed in the following way:

```
q <- p.     // q if p (i.e., if p then q)
r <- !p.    // r if not p (i.e., if not p then r).
```

Applying that rewrite for the current case, we finalize the computation by the following two rules:

```
deathAgeOf[p] = n - 1 <-
    approxDeathAgeOf[p] = n,
    hasDeathdayBeforeBirthday(p).
// person p has an age at death equal to n-1
// if p's approximate death age is equal to
// n, and p's deathday is before p's birthday

deathAgeOf[p] = n <-
    approxDeathAgeOf[p] = n,
    !hasDeathdayBeforeBirthday(p).
// person p has an age at death equal to n
// if p's approximate death age is equal to
// n, and it is not the case that p's deathday is
// before p's birthday.
```

Q5: The rules and constraints we have developed to this point in the exercise are available in file `birthDeath.logic`. The corresponding data for the British monarchs is available in the file `birthDeathData.logic`. Use `lb` to add the former and execute the latter. Then construct an `lb` query to display the age-at-death values. The answers can be seen in file `Q5Answer.logic`.

PART 4: REIGNS AND ANCESTRY

To complete this exercise, you are asked to answer the following questions on your own.

Q6a: Declare the predicates `reignStartOf[m] = d` and `reignEndOf[m] = d` to store facts about the start and end of monarch reigns.

Q6b: Add a constraint to ensure that each monarch started some reign.

Q6c: Add a constraint to ensure that no reign ended before it began.

Q6d: Add a single constraint to ensure both that no dead monarch reigned before he/she was born and that no monarch reigned after he/she died. The answers can be found in file `Q6Answer.logic`. The data from the table itself can be found in the file `reignData.logic`.

Q7a: Add a rule to derive the predicate `daughterOf[m1]=m2` to indicate that monarch `m2` is a daughter of monarch `m1`. Note that in a wider domain where a monarch might have multiple daughters who are monarchs, a functional predicate would not work.

Q7b: Add a rule to derive the predicate `isGrandParentOf(m1,m2)` to indicate that monarch `m1` is a grandparent of monarch `m2`.

Q7c: Enter a query to print out the `isGrandParentOf` predicate. The answers can be found in file `Q7Answer.logic`.

Q8: Write a query to list each monarch who is the parent of at least two monarchs. The answer can be found in file `Q8Answer.logic`.

Q9: Add a rule to derive the predicate `isFirst(m)` indicating who is the first monarch. After installing, query this predicate to display its result. *Hint:* LogiQL does not allow rule bodies to include negated conjunctions if one of the conjuncts is the anonymous variable. So first declare the rule `isLater(m)` to derive the monarchs who reigned later (i.e., after the first monarch), then use that rule to help you derive `isFirst(m)`. The answer can be found in file `Q9Answer.logic`.

ANSWERS TO EXERCISES

Answers to Exercise 1:

1. h—provided by the built-in math library
2. c—file predicates import data from external files
3. f—how person entities are uniquely referred to
4. e—a relation between car entities and person entities
5. d—each general had exactly one age at death; hence, a functional predicate is appropriate
6. a—"the Detroit National Football league team" is an implicit way of describing a team entity
7. b—one entity for each state
8. g—a property of each president; note that some presidents were elected more than once

Answer to Exercise 2:

```
House(h), hasHouseName(h:s) -> string(s).
/* House is an entity type, and hasHouseName is a
   refmode predicate for it. House names are
   represented in the computer as strings. */
```

Answer to Exercise 3:

```
+House("Stuart").  // "Stuart" is the name of a house
+House("Hanover").  // "Hanover" is the name of a
    house
+House("Saxe-Coburg and Gotha").
// "Saxe-Coburg and Gotha" is the name of a house
+House("Windsor").  // "Windsor" is the name of a
    house.
```

Answer to Exercise 4A:

Edward VIII had seven given names!

Answer to Exercise 4B:

`"George"` (the given name George is shared by seven monarchs).

Answer to Exercise 5A:

```
Monarch(m) -> isOfHouse(m, _).
// If m is a monarch then m is of some house.
```

Answer to Exercise 5B:

```
isOfHouse(m, h1), isOfHouse(m, h2) -> h1 = h2.
// Each monarch belongs to at most one house.
```

Answer to Exercise 6A:

```
Gender(g), hasGenderCode(g:gc) -> string(gc).
// g is a gender, identified by its gender code.
// Gender codes are stored as strings.
```

Answer to Exercise 6B:

```
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
// If some gender has a gender code
// then that code is either "M" or "F".
```

This kind of constraint is called a **value constraint** because it constrains the possible values used for the specified role. Here the anonymous variable is used to indicate any arbitrary gender. Note the use of the semicolon (';') for the *inclusive-or* operator.

Answer to Exercise 6C:

```
genderOf[m] = g -> Monarch(m), Gender(g).
// If m has gender g, then m is a monarch and g is a
// gender.
// Each monarch has at most one gender.
```

Answer to Exercise 6D:

```
+Monarch(m), +hasMonarchName(m:"Anne"), +Gender(g),
+hasGenderCode(g:"F"), +genderOf[m] = g.
// "Anne" is a monarchName for Monarch m.
// "F" is a genderCode for a Gender g.
// Monarch m has gender g.
```

Answer to Exercise 6E:

```
+genderOf["George I"] = "M".
// George I has gender with genderCode "M".
```

Answer to Exercise 7A:

```
+parentOf["George II"] = "George I".
+parentOf["George IV"] = "George III".
+parentOf["William IV"] = "George III".
+parentOf["Edward VII"] = "Victoria".
+parentOf["George V"] = "Edward VII".
+parentOf["Edward VIII"] = "George V".
+parentOf["George VI"] = "George V".
+parentOf["Elizabeth II"] = "George VI".
```

Answer to Exercise 7B:

```
isFemale(m) <- genderOf[m] = "F".
// m is female if m has the gender with gender code
// "F".
```

Answer to Exercise 7C:

```
fatherOf[m1] = m2 -> Monarch(m1), Monarch(m2).
fatherOf[m1] = m2 <- parentOf[m1] = m2, isMale(m2).
// The father of m1 is m2 if the parent of m1 is m2
// and m2 is male.
```

Answer to Exercise 7D:

```
isSisterOf(m1, m2) -> Monarch(m1), Monarch(m2).
isSisterOf(m1, m2) <- isSiblingOf(m1, m2),
    isFemale(m1).
// m1 is a sister of m2 if m1 is a sibling of m2,
// and m1 is female.
```

Answer to Exercise 7E:

```
hasNoMonarchSibling(m) -> Monarch(m).
hasNoMonarchSibling(m) <- Monarch(m),
   !isSiblingOf(m, _).
// m has no monarch as a sibling if
// m is a monarch and it is not the case that
// m is the sibling of another monarch.
```

Why do not we also have to say that `!isSiblingOf(_, m)`?

The following monarchs have no monarchs as siblings:

Elizabeth II

George V

Edward VII

Victoria

George III

George II

George I

Why isn't Anne on this list? How would you fix the problem?

Answer to Exercise 7F:

```
"George IV", "William IV"
"Edward VIII", "George VI"
"William IV", "George IV"
"George VI", "Edward VIII"
```

Answer to Exercise 8A:

```
lb create ws
#  Create a new workspace with name "ws".
```

Answer to Exercise 8B:

```
lb addblock -f base.logic ws
#  Add the program in file base.logic to the
#  workspace ws.
```

Answer to Exercise 8C:

```
lb exec -f baseData.logic ws
#  Add facts from the file baseData.logic to the
#  workspace ws.
```

Answer to Exercise 8D:

```
lb print ws House
# Print out the current contents of the predicate
# named House in the workspace ws.
```

Results:

```
Windsor
Saxe-Coburg and Gotha
Hanover
Stuart
```

Answer to Exercise 8E:

```
lb exec ws '_(m, h) <- isFemale(m), houseOf[m] = h.'
# Retrieve names and houses of all female monarchs
# from workspace ws.
```

Results:

```
Elizabeth II, Windsor
Victoria, Hanover
Anne, Stuart
```

Answer to Exercise 8F:

```
lb exec ws '
_(m) <-
     (hasGivenName(m, "George") ; hasGivenName(m,
     "William")),
     !hasGivenName(m, "Albert").
'
# Retrieve names of monarchs with given names "George"
# or "William" but not "Albert" from workspace ws.
```

Note that the parentheses are required to group the two atoms in the disjunction.

Results:

```
William IV
George IV
```

```
  George III
  George II
  George I
```

Answer to Exercise 8G:

```
lb exec ws '
_(m) <-
     hasGivenName(m, n1),
     hasGivenName(m, n2),
     hasGivenName(m, n3),
     n1 ! = n2, n1 ! = n3, n2 ! = n3.
'
      #  Using the workspace ws, retrieve the names of
      #  monarchs with at least three given names.
```

Results:

```
      Elizabeth II
      George VI
      Edward VIII
      George V
      George IV
      George III
```

What would happen if the tests on the last line of the query were left off?