CHAPTER 2

Intermediate Aspects

CONTENTS

This chapter builds on the basic concepts and syntax of LogiQL considered in the previous chapter and introduces some more advanced features of the language. Although the British monarchy remains the primary domain from which examples and exercises are constructed, we begin to introduce other, more business-oriented domains to demonstrate the breadth of applicability of LogiQL.

The first unit considers inverse-functional predicates, which are binary predicates where the first argument is a function of the second argument. The second unit discusses predicates that have more than two arguments and also surveys the various kinds of numeric datatypes available in LogiQL. The next unit covers some constraints that apply to two or more predicate arguments. We then learn how to use subtyping, where a type is contained in a larger type (e.g., `Woman` is a subtype of `Person`), as well as some simple subset constraints. The following unit then examines recursion, one of LogiQL's most powerful features, and discusses constraints that often apply to predicates used in recursive rules. The final unit introduces two of LogiQL's aggregation functions (`count` and `total`), useful for computing properties of a set of facts in the database. The consolidation exercise gives you an opportunity to test your mastery of the new concepts and syntax considered in the chapter.

UNIT 2.1: INVERSE-FUNCTIONAL PREDICATES

Table 2.1 repeats some details about British monarchs discussed in Chapter 1. The `Monarch`, `House`, and `Gender` entity types were declared using the unary predicates `Monarch(m)`, `House(h)`, and `Gender(g)`, and the refmode predicates `hasMonarchName(m:s)`, `hasHouseName(m:s)`, and `hasGenderCode(g:gc)` were declared to provide a natural way to refer to their instances. The facts about their given names, royal houses, and genders

TABLE 2.1 Facts about British Monarchs

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

Note: M, male; F, female.

TABLE 2.2 Gender Codes and Names

Gender	
Code	**Name**
F	Female
M	Male

were captured using the many-to-many predicate `hasGivenName(m,gn)`, and the many-to-one predicates `houseOf[m]=h` and `genderOf[m]=g`.

This unit concerns inverse-functional predicates. As an example, let's revisit our modeling of genders via gender *codes*. Suppose now that we wish to also store gender *names*, as indicated in Table 2.2, so that these can be displayed to users unfamiliar with the codes. This is a very simple example of a two-column lookup table. Lookup tables are often used to store lists of codes and/or names for countries, states, currencies, and so on.

Given our decision to standardly refer to genders by their codes, we may verbalize the association between entries in the two columns:

The gender with gender code "F" has the gender name "Female."

The gender with gender code "M" has the gender name "Male."

To store these facts, we declare the `genderNameOf` predicate. Because each gender has at most one name (i.e., gender name is a function of gender), we use LogiQL's functional notation.

```
genderNameOf[g] = gn -> Gender(g), string(gn).
// If g has gender name gn
// then g is a gender and gn is a string.
// Each gender has at most one gender name.
```

Note also that each gender name relates to at most one gender (i.e., gender is a function of gender name). So the genderNameOf predicate is also functional in the inverse direction (from right argument to left argument). Hence, the genderNameOf predicate is said to be an **inverse-functional predicate**. As there is no special notation for this in LogiQL, we declare this inverse-functional constraint as follows:

```
genderNameOf[g1] = gn, genderNameOf[g2] = gn ->
    g1 = g2.
// For each gender name, at most one gender has that
// gender name.
```

The binary predicate genderNameOf has two roles, one played by gender instances (e.g., g1) and one by gender name instances (gn). The functional constraint that each *gender* occurs at most once in the population of the genderNameOf predicate is said to be a **uniqueness constraint** on the gender role. In this case, it is a *simple* uniqueness constraint because only one role is involved. Similarly, the inverse-functional constraint that each *gender name* occurs at most once in the population of the genderNameOf predicate is a simple uniqueness constraint on the gender name role. Because the genderNameOf predicate is both functional and inverse-functional, it is said to be a **one-to-one predicate**.

An additional constraint is that the genderNameOf predicate is mandatory for Gender, that is,

```
Gender(g) -> genderNameOf[g] = _.
// If g is a gender then it has some gender name.
```

A 1:1 relationship that is mandatory for all instances of its first argument is said to be *injective*. Hence, the genderNameOf predicate is injective. Refmode predicates are always understood to be injective in this sense, so the colon used to separate arguments of refmode predicates constrains them to be mandatory and 1:1. This colon syntax is used only for refmode predicates, so if a non-refmode predicate is injective, this must be declared explicitly, as shown above for genderNameOf.

Just as a 1:1 predicate can have an inverse, the inverse of a **many-to-one predicate** is said to be a **one-to-many predicate** (or *1:n predicate*). For example, `isMotherOf(p1,p2)` is a 1:*n* predicate because a single person may be the mother of many persons, even though each person has at most one mother. Both 1:*n* and 1:1 predicates are inverse-functional predicates, and a 1:1 predicate is also functional. If you declare a 1:*n* predicate, you need to declare its uniqueness constraint separately. For example,

```
isMotherOf(p1, p2) -> Person(p1), Person(p2).
// If p1 is mother of p2, then p1 and p2 are persons.

isMotherOf(p1, p), isMotherOf(p2, p) -> p1 = p2.
// Each person has at most one mother.
```

Note that if you have a 1:*n* predicate, in most cases, it's better to replace it by its inverse, which you can declare as an *n*:1 predicate simply by using the functional notation. For example,

```
motherOf[p1] = p2 -> Person(p1), Person(p2).
// If p1 has mother p2, then p1 and p2 are persons.
// Each person has at most one mother.
```

Table 2.3 summarizes the four kinds of binary predicates we have discussed. Each type of predicate relates instances of the first argument role with instances of the second argument role. The rows in the Table 2.3 indicate how many instances of the first argument may be related to a number of instances of the second type. For example, an *n:1* predicate may contain multiple (*n*) facts with different instances of the first argument pertaining to a single (*1*) instance of the second argument. For the `motherOf` predicate, multiple children (`p1`) may have the same mother (`p2`).

TABLE 2.3 Types of Predicate Cardinalities

Predicate Type	**Example**	**Functional**	**Inverse-Functional Constraint Needed**
m:*n*	`hasGivenName(p,gn)`	No	No
n:1	`motherOf[p1] = p2`	Yes	No
1:1	`genderNameOf[g] = gn`	Yes	Yes
1:*n*	`isMotherOf(p1,p2)`	No	Yes

Tip: If you have a 1:*n* predicate, consider replacing it by, or deriving it from, its inverse *n*:1 predicate.

Exercise 1A: Declare the predicate `isFatherOf` over person pairs and constrain it to be inverse-functional.

Exercise 1B: Declare the functional predicate `fatherOf`, as well as a rule to derive `isFatherOf` from it.

UNIT 2.2: *N*-ARY PREDICATES

Consider Table 2.4 as discussed in Chapter 1. Earlier, we modeled the relationship between monarchs and their given names with the *m:n* predicate `hasGivenName(m,gn)`. This is fine, so long as we are not interested in recording the *order* of a monarch's given names. Now suppose we are interested in this order (e.g., we want to know that George I had "George" as his first given name and "Louis" as his second given name). How would you express this?

One way to model this requirement is to use multiple given name predicates, one for each position (Table 2.4), for example,

```
firstGivenNameOf[m] = gn
secondGivenNameOf[m] = gn
thirdGivenNameOf[m] = gn
...
```

TABLE 2.4 Given Names for 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

This is reasonable if a monarch may have just a few given names. However, Edward VIII (the only British monarch so far to abdicate) had seven given names. Because of this large number and the problem of setting a limit on how many given names a monarch may be assigned, it's far better to use a predicate that includes a position to indicate the place of the given name. For example, we could verbalize the facts about George I's given names informally; thus,

George I, in position 1, has the given name "George."

George I, in position 2, has the given name "Louis."

To indicate the position, we use the LogiQL integer (abbreviated as "`int`") primitive datatype. It is specified in the form `int(p)`, where `p` is the name of the variable holding the position. Hence, we can now declare the relevant predicate as shown in the following code:

```
inPositionHasGivenName[m, n] = gn -> Monarch(m),
    int(n), string(gn).
/* If m in position n has given name gn, then m is a
   monarch, n is an 8 bit unsigned integer, and gn is
   a string. For each monarch and position, there is
   at most one given name. */
```

The `inPositionHasGivenName` predicate has three arguments (two within the square brackets and one to the right of the equals sign), so it is an example of a *ternary predicate*. In general, if a predicate has *n* arguments, and *n* > 2, we call it an *n-ary predicate*.

Note the use of the functional notation, with the variables *m* and *n* in square brackets, to express the *compound* uniqueness constraint that for a given monarch and position in the monarch's given name list there is at most one given name. That is, the combination of monarch and position functionally determines the given name. Analogous to the notion of primary key in a relational database table, the arguments in square brackets comprise the keyspace of the functional predicate (or *compound key*). For a predicate that is not functional (e.g., a many-to-many predicate), the collection of all of its arguments also comprise a keyspace, since each instance in the population of the predicate is unique. Such predicates are called *all-key*.

To ensure that given name positions start at 1 rather than 0, we can declare the following value constraint on the position argument. Notice the use of anonymous variables for the other arguments.

```
inPositionHasGivenName[_, n] = _ -> n > 0.
// If something in position n has some given name
// then n is greater than 0.
```

Note that in this example, there were two uses of anonymous variables—one for monarchs and one for given names. Be aware that even though the same symbol (' _ ') was used for both, these represent different variables.

If you look at the table of monarch names, you should see that there is another uniqueness constraint on this ternary predicate. For each monarch, each given name occurs in at most one position. For example, no monarch would be given the names "George, George." This compound uniqueness constraint needs to be declared separately, because the functional syntax can be used for only one choice of keyspace. The unit exercise challenges you to code this constraint in LogiQL.

Just as a binary predicate may be *m:n* and hence all-key, it is possible that an *n*-ary predicate is all-key. Here is an example of a many-to-many-to-many, ternary predicate taken from a business domain—keeping track of retail products. In the example, we declare the following predicate to record what products are available at what stores in what seasons:

```
productAvailability(p, st, se) -> Product(p),
    Store(st), Season(se).
/* If p is available for st during se, then p is a
   product, st is a store and s is a season. */
```

The keyspace of an *n*-ary functional predicate typically has *n*–1 arguments, with the final argument being a function of the previous arguments. As an example of a functional predicate to record the number of units sold for a given product in a given store in a given season, we might declare the following predicate (assuming `Product`, `Store`, and `Season` are also declared):

```
productStoreSeasonUnitsSold[p, st, se] = nrSold ->
    Product(p), Store(st), Season(se), int(nrSold).
// If p is sold in st during se in amount nrSold, then
// p is a product, st is a store, se is a season, and
// nrSold is an int.
```

Typically, any asserted predicate should be *atomic* (i.e., not equivalent to a conjunction of smaller predicates). For this reason, it is rare to use predicates with more than five arguments. By restricting asserted predicates to be atomic, we capture the information as simply as possible, avoid the need to deal with missing or inapplicable values, and facilitate subsequent changes to the model.

Tip: Make sure that your asserted predicates are all atomic.

Exercise 2A: Modify the value constraint given above for positions in given name lists (`inPositionHasGivenName`) to ensure that each position number is in the range 1..9 (i.e., at least 1 and at most 9).

Exercise 2B: Code the constraint that for each monarch, each given name occurs in at most one position.

Exercise 2C: Write a derivation rule, `sharesGivenName(m1, m2)`, that derives pairs of `Monarchs` that share a given name.

Exercise 2D: Your solution to **Exercise 2C** likely listed both the pair `("George I", "George II")` and the pair `("George II", "George I")`. Write another rule, `sharesGivenName2(m1,m2)`, that derives pairs of `Monarchs` that share a given name without including this form of redundancy. *Hint:* Only include a `Monarch` pair if the monarch name of the first monarch is alphabetically less than the monarch name of the second.

UNIT 2.3: INCLUSIVE-OR AND EXTERNAL UNIQUENESS CONSTRAINTS

A previous unit considered the genders of the British monarchs as presented in Table 2.5, where the gender codes "M" and "F" denote male and female genders. The following code shows the way we model this information in LogiQL, using a binary, functional `genderOf` predicate. We include a value constraint to limit the possible gender codes and a simple mandatory constraint to ensure that each monarch has a gender:

```
Monarch(m), hasMonarchName(m:n) -> string(n).
Gender(g), hasGenderCode(g:gc) -> string(gc).
genderOf[m] = g -> Monarch(m), Gender(g).
genderOf[_] = g -> g = "M" ; g = "F".
Monarch(m) -> genderOf[m] = _.
```

TABLE 2.5 Genders of British Monarchs

Monarch	**Gender**
Anne	F
George I	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

Note: M, male; F, female.

As an alternative way to model the information, we briefly considered using the unary `isMale` and `isFemale` predicates:

```
Monarch(m), hasMonarchName(m:n) -> string(n).
isMale(m) -> Monarch(m).
isFemale(m) -> Monarch(m).
isMale(m) -> !isFemale(m).
// No monarch is both male and female.
```

The exclusion constraint on the fourth line captures the functional nature of the `genderOf` predicate (each monarch has at most one gender), but to ensure that each monarch has a gender, we need to add the following **inclusive-or constraint** that each monarch is either male or female:

```
Monarch(m) -> isMale(m) ; isFemale(m).
// Each monarch is either male or female.
```

In this example, the `isMale` and `isFemale` disjuncts are also mutually exclusive. The combination of the inclusive-or and exclusion constraints is known as an *exclusive-or* constraint. Unlike Latin, LogiQL does not have two varieties of the *or* operator, one inclusive and one exclusive. So you need to code an exclusive-or constraint as two constraints, one for the inclusive-or aspect and one for the exclusive aspect, as done above.

In the second approach just described, the `isMale` and `isFemale` predicates were asserted, not derived. In this case, any constraints on the predicates need to be explicitly declared. However, suppose that in addition to using the genderOf predicate to assert the gender facts, we also derive the isMale and isFemale predicates, using the following derivation rules:

```
isMale(m) <- genderOf[m] = "M".
isFemale(m) <- genderOf[m] = "F".
```

In this case, there is no need to assert the inclusive-or and exclusion constraints on the isMale and isFemale predicates, because these constraints are implied by the constraints declared for the genderOf predicate.

Inclusive-or constraints may apply to two or more roles of predicates of any arity, so long as the roles are played by instances of compatible types. For example, suppose that in our business domain, each product *p* included in a sale must satisfy at least one of the following requirements: *p* is a discontinued product; *p* is a loss leader; or *p* was nominated by a marketing analyst. Assuming SaleProduct and MarketingAnalyst are already declared, we could code this situation as follows:

```
isDiscontinued(p) -> SaleProduct(p).
isLossLeader(p) -> SaleProduct(p).
wasNominatedBy[p] = m -> SaleProduct(p),
    MarketingAnalyst(m).
SaleProduct(p) ->
    isDiscontinued(p) ; isLossLeader(p) ;
    wasNominatedBy[p] = _.
// Each sale product is discontinued or is a loss
// leader or was nominated by a marketing analyst.
```

For simplicity, the only products of interest in this example are sales products.

Now consider Table 2.6, an extract from a table used by a particular business to record the major cities where the company has offices. In this business domain, cities are primarily identified by city numbers, since these provide a simple, rigid identifier that remains valid even if the city changes its name (e.g., consider Constantinople and St. Petersburg). States are identified using state codes (e.g., "ME" for Maine and "OR" for Oregon). Since many users of the information system might not know the city numbers, the combination of city name and state provides an alternative reference scheme to enable them to easily refer to the cities.

TABLE 2.6 Locations of Company Offices

City Number	City Name	State
1	Portland	Maine
2	Portland	Oregon
3	Eugene	Oregon
…	…	…

The basic structure (*schema*) and data for this example may be coded as shown below, using techniques already discussed:

```
// Schema
City(c), hasCityNr(c:n) -> int(n).
State(s), hasStateCode(s:sc) -> string(sc).
cityNameOf[c] = cn -> City(c), string(cn).
stateOf[c] = s -> City(c), State(s).
City(c) -> cityNameOf[c] = _.
City(c) -> stateOf[c] = _.

// Data
+cityNameOf[1] = "Portland", +stateOf[1] = "ME".
+cityNameOf[2] = "Portland", +stateOf[2] = "OR".
+cityNameOf[3] = "Eugene", +stateOf[3] = "OR".
```

What is missing from this code is a constraint to ensure that each combination of city name and state refers to only one city. This constraint may be coded as follows:

```
cityNameOf[c1] = cn, stateOf[c1] = s,
    cityNameOf[c2] = cn, stateOf[c2] = s ->
    c1 = c2.
// Each combination of city name and state refers to
// at most one city.
```

This is called an *external* uniqueness constraint, since the uniqueness applies to roles from multiple predicates rather than being internal to a single predicate. As a check that the constraint is enforced, if you try to execute the following fact assertions on the above schema, you will get a runtime error indicating that a constraint has been violated:

```
+cityNameOf[4] = "Seattle", +state[4] = "WA". // Error!
+cityNameOf[5] = "Seattle", +state[5] = "WA". // Error!
```

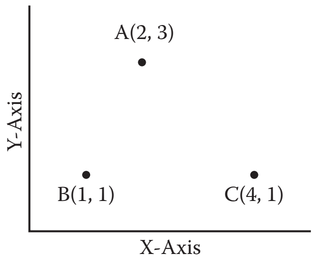

FIGURE 2.1 Company location grid.

Constraint violation errors are detected only after the compiler has checked that there are no syntax errors, such as writing `"stateOf(c1) = s"` in the program code or omitting the "+" in a data assertion.

Tip: Add relevant constraints to predicates to prevent them being populated with data that is inconsistent with the application domain.

Exercise 3A: A given company identifies its employees by employee numbers but also requires each employee to have either an identifying Social Security Number or an identifying passport number. Express this situation in LogiQL, using `strings` to store the identifying numbers.

Exercise 3B: The neighborhoods in which a company has stores are identified by (single-character) names. Each store can also be referenced by the combination of its *x* and *y* map coordinates, as shown in Figure 2.1. Express this situation in LogiQL using integers for the coordinates, and issue a query to list the label and coordinates of each store.

UNIT 2.4: SUBTYPING AND SUBSETTING

Types are an important concept in most programming languages, and LogiQL is no exception. The built-in (primitive) datatypes include boolean, numbers of various sorts (`int`, `float`, and `decimal`), `string`, and `datetime`. In addition, as described in Chapter 1, programmers can declare their own entity types. This unit discusses an additional, powerful feature of types called *subtyping*. Subtyping is a way of indicating that a set of the entities of one type, the **subtype**, must be a subset of the entities of another type, the **supertype**. To illustrate the possibilities, we consider `King` and `Queen` as subtypes of `Monarch` in the British royalty as presented in Table 2.7.

TABLE 2.7 Another Representation of Monarch Genders

Monarch	**King**	**Queen**
Anne		✓
George I	✓	
George II	✓	
George III	✓	
George IV	✓	
William IV	✓	
Victoria		✓
Edward VII	✓	
George V	✓	
Edward VIII	✓	
George VI	✓	
Elizabeth II		✓

Subtypes

Table 2.7 lists the British monarchs, indicating the kings and queens by a check mark (✓) in the relevant column. For now, let us assume that the `genderOf` predicate discussed in earlier units is not declared. If we wish to think of kings and queens as entities, we may declare the entity types `King` and `Queen` as follows:

```
Monarch(m), hasMonarchName(m:n) -> string(n).
King(m) -> Monarch(m).
// If m is a king, then m is a monarch.
lang:isEntity[`King] = true.
// King is an entity type.
Queen(m) -> Monarch(m).
// If m is a queen, then m is a monarch.
lang:isEntity[`Queen] = true.
// Queen is an entity type.
```

Here, the constraints indicate that each king and queen is also a monarch, making `King` and `Queen` subtypes of `Monarch`. Equivalently, `Monarch` is a supertype of `King` and `Queen`. The **metapredicate** `lang:isEntity` is used to declare that the predicate in parentheses after the grave accent character (‘`’) is an entity predicate.

The `King` and `Queen` subtypes inherit the properties of their supertype `Monarch`, including its refmode identification scheme.

For example, we may assert that Anne is a queen and that George I is a king as follows:

```
+Queen("Anne"), +King("George I").
```

Because of the declarations for `King` and `Queen`, the compiler is able to infer that Anne and George I are also monarchs—that is, `Monarch("Anne")` and `Monarch("George I")`. There is no need to include `+Monarch("Anne")` and `+Monarch("George I")` assertions. Here, `King` and `Queen` are said to be *asserted subtypes* because we simply assert their instances rather than derive them from other facts.

To complete the example, we should add the following constraints to ensure that `King` and `Queen` form a partition (disjoint union) of `Monarch`:

```
Monarch(m) -> King(m) ; Queen(m).
// Each monarch is a king or a queen.
King(m) -> !Queen(m).
// No king is a queen.
```

Because they have not been declared supertypes of other types, `King` and `Queen` are said to be *leaf* subtypes. Leaf predicates without `lang:isEntity` declarations are treated as simple property predicates rather than entity predicates. In such situations, if you really do not want to think of them as entity predicates, it's better to name the predicates using verb phrases (e.g., `isaKing`, `isaQueen`) rather than noun phrases.

Now suppose that instead of the above approach of asserting subtype entities, we chose to explicitly indicate the gender of each monarch using the `genderOf` predicate and to define kings and queens in terms of their gender. Table 2.8 and the code that follows it represent this new situation.

```
Monarch(m), hasMonarchName(m:n) -> string(n).
Gender(g), hasGenderCode(g:gc) -> string(gc).
genderOf[m] = g -> Monarch(m), Gender(g).
Monarch(m) -> genderOf[m] = _.
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
King(m) -> Monarch(m).
King(m) <- genderOf[m] = "M".
Queen(m) -> Monarch(m).
Queen(m) <- genderOf[m] = "F".
```

TABLE 2.8 Combined Gender Presentation

Monarch	Gender	King	Queen
Anne	F		✓
George I	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		✓

Note: M, male; F, female.

In this case, you might expect `King` and `Queen` to be subtype entities that had been derived from `Monarch`. However, the absence of the `lang:isEntity` declarations for `King` and `Queen` requires LogiQL to treat them as property predicates rather than entity predicates (which we should really have emphasized by beginning their names with lowercase letters). Moreover, note that there is no need to declare the inclusive-or and exclusion constraints used above to ensure that `King` and `Queen` partition `Monarch`, because the `King` and `Queen` predicates are derived, and these partition constraints are implied by the functional, mandatory, and value constraints on the predicates used to define them.

Alternatively, if you explicitly declare the types of the `King` and `Queen` predicates, then you may declare them to be entity predicates as shown below. In this case, `King` and `Queen` may be properly viewed as *derived subtypes* because their instances are determined using rules rather than being simply asserted:

```
Monarch(m), hasMonarchName(m:n) -> string(n).
Gender(g), hasGenderCode(g:gc) -> string(gc).
genderOf[m] = g -> Monarch(m), Gender(g).
Monarch(m) -> genderOf[m] = _.
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
King(m) -> Monarch(m). // Type declaration
  lang:isEntity[`King] = true.
King(m) <- genderOf[m] = "M".
```

```
Queen(m) -> Monarch(m). // Type declaration
lang:entity[`Queen] = true.
Queen(m) <- genderOf[m] = "F".
```

The above derivation rules for the `King` and `Queen` predicates are trivially based purely on gender because in our current application domain the only people are monarchs. Suppose we now expand our universe of discourse to include any royal family member of interest, where each can be identified by a "royal name" such as "Elizabeth II" or "Prince William." Even though, historically, there has been more than one royal family member named "Prince William," let us assume there is only one of these of interest in our domain, in this case, the Prince William of Wales born in 1982.

We could now record the gender of each royalty, assert who are monarchs, and derive who is a king and who is a queen using the following schema. Here we use *royalty* in the singular sense, and use the variable `p` in the sense of *person*:

```
Royalty(p), hasRoyalName(p:n) -> string(n).
Gender(g), hasGenderCode(g:gc) -> string(gc).
genderOf[p] = g -> Royalty(p), Gender(g).
Royalty(p) -> genderOf[p] = _.
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
Monarch(p) -> Royalty(p). // Asserted subtype
King(p) -> Monarch(p).
lang:isEntity[`King] = true.
King(p) <- Monarch(p), genderOf[p] = "M".
// Derived subtype
Queen(p) -> Monarch(p).
lang:isEntity[`Queen] = true.
Queen(p) <- Monarch(p), genderOf[p] = "F".
// Derived subtype
```

In this code, `Monarch` is an asserted subtype of `Royalty`, while `King` and `Queen` are derived subtypes of `Monarch`. Because subtyping is **transitive**, this means that `King` and `Queen` are also (indirect) subtypes of `Royalty`. If you run this program with the following data, the compiler will infer that Anne is a queen, George I is a king, and all three persons are royalty:

```
+genderOf["Anne"] = "F", +Monarch("Anne").
+genderOf["George I"] = "M", +Monarch("George I").
+genderOf["Prince William"] = "M".
```

Note that we did not bother declaring that `Monarch` is an entity predicate, because `Monarch` is no longer a leaf subtype. The compiler is able to infer that `Monarch` is an entity predicate because it appears on the right-hand side of the type declaration for `King` (or `Queen`), which have themselves been explicitly declared to be entity types. Nevertheless, because `King` and `Queen` are leaf subtypes, explicit entity declarations are required.

The subtyping examples discussed so far are all examples of *single inheritance*, where each subtype has at most one direct supertype. If a subtype has more than one direct supertype, this is called *multiple inheritance*. For example, we might define `Queen` and `DeadMonarch` as separate but overlapping subtypes of `Monarch`, and define `DeadQueen` as a subtype of both `Queen` and `DeadMonarch`. Be aware that LogiQL does not properly support multiple inheritance, and you can easily generate inconsistent results if you try to use it.

Subset Constraints

Subtyping constrains one entity type to be a subtype of another. Similarly, for property predicates, we can constrain the population of one property predicate to always be a subset of the population of another property predicate. This is known as a **subset constraint**. As a simple example of a subset constraint between two property predicates, consider the following code. Here `hasGardenCenter` and `sellsLawnMowers` are property predicates of the `Store` entity type. The final line of code, just above the comment, declares a subset constraint from the `hasGardenCenter` predicate to the `sellsLawnMowers` predicate:

```
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.
```

Here is another example, this time of a subset constraint between role pairs (the `s` and `p` arguments of `purchased` and `storeProductNr-Sold`), to ensure that stores sell only those products that they have already purchased:

```
Store(s), hasStoreNr(s:n) -> int(n).
Product(p), hasProductCode(p:pc) -> string(pc).
```

```
purchased(s, p) -> Store(s), Product(p).
storeProductNrSold[s, p] = n -> Store(s), Product(p),
    int(n).
storeProductNrSold[s, p] = _ -> purchased(s, p).
// If store s sold product p in some quantity
// then store s purchased that product p.
```

Tip: Use a noun phrase to name an entity predicate, and use a verb phrase to name a unary property predicate.

Tip: If a leaf subtype is intended to be an entity type, declare this explicitly using a `lang:isEntity` declaration.

Tip: Avoid multiple inheritance.

Tip: Although the LogiQL subtyping mechanism enables you to define new entity types, it does not allow you to define subtypes of primitive types, such as defining a type of string comprising the names of the months. To do this, you should use subset constraints instead.

Exercise 4A: The following program is used to record information about companies. Some companies (dealers) sell products, while others provide services. Extend the program to include derived subtypes for `Dealer` (companies that sell products), `NonDealer` (all other companies), and `IPhoneDealer` (companies that sell iPhones). Then install the program, execute the data, and query the `Store`, `Dealer`, `NonDealer`, and `IPhoneDealer` predicates:

```
// Schema
Company(c), hasCompanyName(c:n) -> string(n).
Product(p), hasProductName(p:n) -> string(n).
carries(c, p) -> Company(s), Product(p).

// Data
+Company("Target"), +Company("KMart"),
+Company("Sears").
+carries("Target", "Blackberry").
+carries("Sears", "IPhone").
```

Exercise 4B: Extend the following program with a subset constraint to ensure that bookstores own the coffee shop(s) that they provide:

```
BookStore(s), hasStoreName(s:n) -> string(n).
CoffeeShop(c), hasShopName(c:n) -> string(n).
provides(s, c) -> BookStore(s), CoffeeShop(c).
owns(s, c) -> BookStore(s), CoffeeShop(c).
```

UNIT 2.5: RECURSIVE RULES AND RING CONSTRAINTS

In Chapter 1, the `isGrandparentOf` predicate was introduced to express the grandparenthood relationship between monarchs in terms of the parenthood relationship. We could have also defined other variants, such as `isGreatGrandparentOf`. What we could not have done with the concepts available at that time was define the more general relationship of *ancestry*.

Recursion

The problem is that if one person is an ancestor of another there may be any number of intermediate parents. Expressing this imprecision requires the more powerful notion of **recursion**. Roughly speaking, recursion is the use of a concept in its own definition. In LogiQL, recursion can be used in derivation rules and constraints.

For generality, we will now consider people other than monarchs, so that ancestorhood is a many-to-many relationship between two persons. Let's see how we could specify an `isAncestorOf` derivation rule using recursion. If we were to verbalize the rule, we might say the following:

> *Person p1 is an ancestor of person p2 if either p1 is a parent of p2 or p1 is a parent of some ancestor of p2.*

Note the two uses of "ancestor" in the above definition. The first gives the term being defined, while the second is part of the definition.

The use of "or" in the above description indicates that the corresponding derivation rule is disjunctive. A disjunctive rule can be expressed directly using the *or* operator (`';'`), or it can be replaced by multiple non-disjunctive rules, one for each disjunct. Let's use the latter approach. The first disjunct in the above ancestor rule can be expressed by a non-recursive rule as follows, assuming that the `isParentOf` *m:n* predicate is defined between two monarchs:

```
isAncestorOf(p1, p2) <- isParentOf(p1, p2).
// If p1 is a parent of p2 then p1 is an ancestor of p2.
```

The second part is nearly as simple once you realize that you can use `isAncestorOf` on the right-hand side:

```
isAncestorOf(p1, p2) <- isParentOf(p1, p3),
    isAncestorOf(p3, p2).
// If p1 is a parent of some p3 and p3 is an ancestor
// of p2 then p1 is an ancestor of p2.
```

The `isAncestorOf` predicate was defined in two parts, which is typical of recursive definitions. The first rule, known as a *basis clause*, deals with the *ground* case where there is no recursion—that is, where ancestorhood is simply parenthood. The second rule, known as a *recursive clause*, derives an ancestorhood fact from a parenthood fact and an ancestorhood derivation that is one step shorter. Repeated applications of this rule eventually spiral into a case where only one step is left, and this case is satisfied by the basis clause.

Any derivation rule that includes the same predicate in both the head and body of the rule is a **recursive rule**. Recursion is a powerful capability, and the elegant and efficient way in which LogiQL supports recursion is one of its most attractive features. In general, when you have a predicate, such as `isParentOf`, that you wish to apply an indefinite number of times to compute a new predicate (`isAncestorOf`), you should use recursion. Such situations compute what is called the **transitive closure** of the original predicate, and rules to compute transitive closures are needed in many applications.

Recursion can also be applied to constraints. We will use the genealogy graph for the current Prince George going back through four generations to illustrate several constraints, including one that is recursive, as shown in Figure 2.2.

In Figure 2.2, each node in the graph denotes a person, using rectangles for males and rounded rectangles for females. Line segments that connect nodes represent biological parenthood relationships, read from left to right. For example, Prince Charles and Princess Diana are the parents of Princes William and Harry.

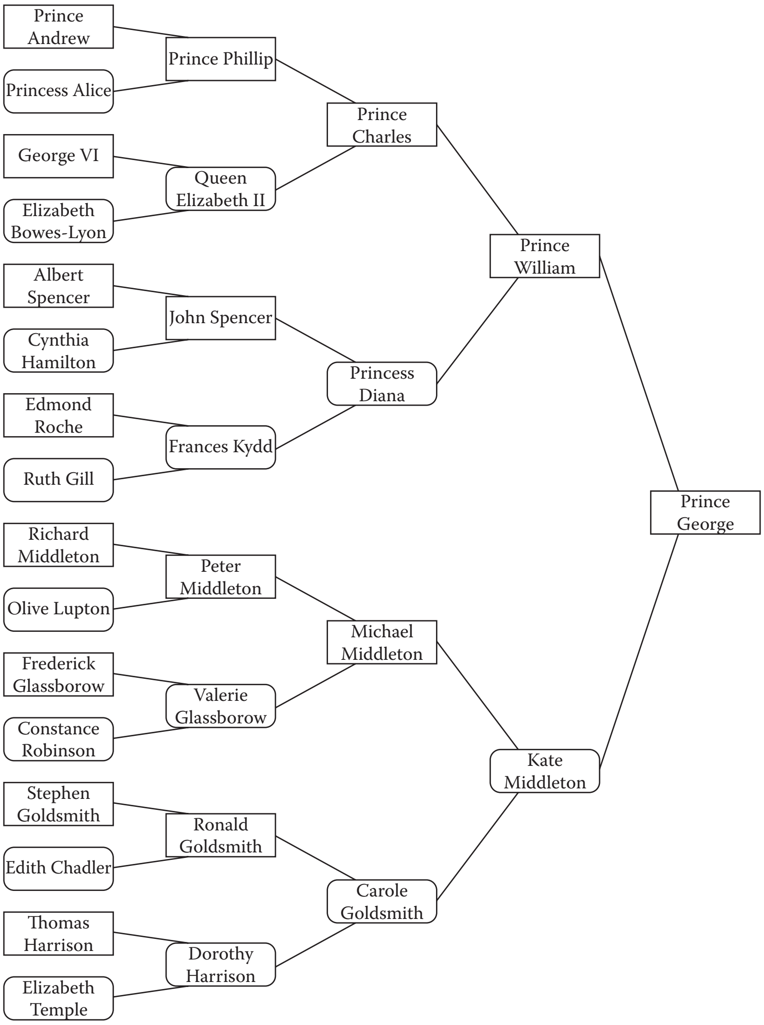

FIGURE 2.2 Genealogy for Prince George.

Treating each of the royalty simply as a person and their identifiers in the graph nodes as person names, we may use the following basic schema to encode the gender and parenthood facts. Because a person may have up to two parents listed, and some have more than one child, the parenthood predicate is many-to-many:

```
Person(p), hasPersonName(p:n) -> string(n).
Gender(g), hasGenderCode(g:gc) -> string(gc).
```

```
genderOf[p] = g -> Person(p), Gender(g).
isParentOf(p1, p2) -> Person(p1), Person(p2).
Person(p) -> genderOf[p] = _.
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
```

Ring Constraints

Notice that both arguments of the `isParentOf` predicate are of type `Person`. A binary predicate whose arguments are of the same type is called a **ring predicate**. Typically, ring predicates need to be constrained in various ways to prevent them being populated with bad data. For example, we should constrain the parenthood predicate to be **irreflexive** (i.e., no instance may participate in the relationship with itself) by adding the following constraint:

```
!isParentOf(p, p).
// No person is a parent of his/her self.
// Note that there is no head nor arrow on this
// constraint.
```

If you add this constraint to the above program, compile it, and try to assert the following data, you will get an error because the second line of data violates the above irreflexive constraint:

```
+genderOf["George V"] = "M".
+isParentOf("George V", "George V"). // Error!
```

The file containing the correct data for the figure is available as `PrincesData.logic`.

Logical constraints on two type-compatible arguments of a predicate are called **ring constraints**. An irreflexive constraint is just one of many kinds of ring constraint. We now discuss some other varieties of ring constraints.

A ring predicate is **asymmetric** if it works in one direction only. For example, the `isParentOf` predicate is asymmetric, and we can constrain it to be so by adding the following *asymmetry constraint*:

```
isParentOf(p1, p2) -> !isParentOf(p2, p1).
// If p1 is a parent of p2, then p2 cannot be a parent
// of p1.
```

For example, if you try to assert that George V is a parent of George VI and also assert that George VI is a parent of George V, you will violate this constraint. If a predicate is asymmetric, it must be irreflexive, so if you include the asymmetric constraint above, there is no need to include the irreflexive constraint given above.

A ring predicate *R* is *intransitive* if and only if, given any *x*, *y*, and *z*, if *R*(*x*, *y*) and *R*(*y*, *z*) are true, then *R*(*x*, *z*) must be false. Assuming no incest, the parenthood relationship is intransitive, and we may constrain it to be so by adding the following *intransitive constraint*:

```
isParentOf(p1, p2), isParentOf(p2, p3) ->
    !isParentOf(p1, p3).
// If p1 is a parent of p2, and p2 is a parent of p3
// then p1 cannot be a parent of p3.
```

Note that if a binary predicate is both functional and irreflexive, it must be intransitive.

A ring predicate *R* is **acyclic** if and only if no object may cycle back to itself by one or more applications of *R*. Ignoring the possibility of reincarnation, the parenthood relationship is acyclic. Since there is no restriction on how many times the *R* predicate is applied, we need to use recursion to express an acyclic constraint.

To ensure that a ring predicate *R* is acyclic, we first recursively derive a predicate *T* that captures all pairs of arguments resulting from one or more applications of *R*. That is, *T* is the transitive closure of *R*. We then constrain *T* to be irreflexive. With our current example, *R* is the `isParentOf` predicate, and *T* is the `isAncestorOf` predicate. Hence, we can specify the acyclic constraint on parenthood by constraining ancestorhood to be irreflexive as follows:

```
!isAncestorOf(p, p).
// No person is an ancestor of himself/herself.
```

If you look back at Figure 2.2, you should be able to see that this prevents any cycles appearing in the graph of parenthood facts (i.e., parenthood is acyclic).

Note that acyclicity implies asymmetry. So if you constrain a predicate to be acyclic there is no need to also declare it to be asymmetric (or irreflexive).

Hard and Soft Constraints

LogiQL constraints are checked at runtime when new data is added to the workspace, either by fact assertion or by derivation from a rule. If a constraint is violated, program execution is stopped, and an error reported to the user. For many types of constraints, this is exactly the behavior that we want. For example, if two facts are asserted, one stating that Queen Anne is a female and one that she is a male, we want to be told immediately, so we can correct the data. Such constraints are called **hard constraints** and indicate what conditions must necessarily hold for the application domain.

Sometimes, however, there are constraints that could be violated in the domain of interest, even though they ought to be obeyed. For example, the above intransitive constraint on parenthood could be violated if incest occurs. These are examples of **soft constraints**. Ideally, we would like violations of soft constraints to be handled differently from how hard constraints are handled. In particular, our programs should be able to detect them, take appropriate action, such as issuing a warning message, and continue processing. Unfortunately, LogiQL does not support this capability. Instead, if we know that parenthood is intransitive in the domain being modeled (as in this royal ancestry domain), then it may be declared as a hard constraint.

Tip: If a predicate has two arguments of the same type, consider which ring constraints apply to it.

Exercise 5A: The ancestry program discussed is accessible as the file `Ancestry.logic`, and the ancestral data depicted in Figure 2.2 is accessible as the file `PrincesData.logic`. Compile the program, execute the data file, and then issue a query to list all the ancestors of Prince William.

Exercise 5B: A typical business application of recursion is a bill-of-materials report. The following schema and data are used to record which products directly contain which other products in what quantities. Since the first two arguments of the direct containment predicate are `Products`, one or more ring constraints might apply. Extend the schema to express the predicate `contains(p1,p2)`, which indicates which product contains which product (directly or indirectly). Use `contains` to constrain direct containment to be acyclic. Then query this predicate to provide information about all instances of direct and indirect containment:

```
// Schema
Product(p), hasProductCode(p:pc) -> string(pc).
directlyContainsIn[p1, p2] = qty -> Product(p1),
    Product(p2), int(qty).

// Data
+directlyContainsIn("A", "B", 1).
+directlyContainsIn("A", "C", 2).
+directlyContainsIn("B", "C", 1).
+directlyContainsIn("B", "D", 1).
+directlyContainsIn("B", "E", 2).
+directlyContainsIn("C", "E", 2).
+directlyContainsIn("C", "F", 2).
```

Exercise 5C: Assuming no incest, the parenthood predicate is not just intransitive, but *strongly* intransitive, so that no person can be a parent of any of his/her non-direct descendants. Code a constraint to ensure that parenthood is strongly intransitive.

UNIT 2.6: THE `count` AND `total` FUNCTIONS

This unit considers the first two of the four most important **aggregation functions** supported by LogiQL: `count`, `total`, `max`, and `min`. Each of these four functions operates on a collection of facts and returns a single value. Aggregation functions are invoked using a special `agg<<…>>` syntax. Hence, rules referring to aggregation functions are called **aggregation rules** to distinguish them from other kinds of derivation rules. Inside the double-angle brackets, a variable is assigned the result of applying the indicated aggregation function to facts satisfying a filtering condition that follows the closing double-angle bracket.

Computing Counts

To invoke the `count` function, the following pattern is used:

```
agg<<n = count()>> condition
// n is the number of instances where condition is true.
```

TABLE 2.9 Genders of British Monarchs

Monarch	Gender
Anne	F
George I	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

Note: M, male; F, female.

As our first example, we'll use the `count` function to derive the number of British monarchs, as listed in Table 2.9. The code below provides the basic schema for this domain, as discussed in Chapter 1:

```
Monarch(m), hasMonarchName(m:n) -> string(n).
Gender(g), hasGenderCode(g:gc) -> string(gc).
genderOf[m] = g -> Monarch(m), Gender(g).
genderOf[_] = g -> g = "M" ; g = "F".
Monarch(m) -> genderOf[m] = _.
```

Let us use the predicate `nrMonarchs[]=n` to denote the number of monarchs. Because this function takes no key arguments, you may also think of it as a variable whose value at any given moment of time is the number of `Monarch` entities. Note that to be able to hold the result of the `count` function, the value argument of `nrMonarchs[]` must be of type `int`. The following code uses the `count` function to derive the number of monarchs:

```
nrMonarchs[] = n -> int(n).
nrMonarchs[] = n <- agg<<n = count()>> Monarch(_).
// nrMonarchs = number of monarchs.
```

The `count` function, written as `count()`, returns the number of instances satisfying the condition specified after the double-angle brackets. In this case it counts the number of facts where some object (denoted by the anonymous variable) is a monarch.

If you install the program with the monarchy data shown above and then print or query the nrMonarchs predicate, you will get 12 as the result.

In the above example, we declared the nrMonarchs predicate along with a derivation rule using it. You may also use the count function in ad hoc queries, using an anonymous predicate ('_[]') for the query result. For example, you could query for the number of kings using the following query string:

```
_[] = n <- agg<<n = count()>> Monarch(m),
    genderOf[m] = "M".
// Return the number of kings (male monarchs).
```

Unlike the earlier use of count, the condition to be satisfied here is a conjunction. Be aware however that disjunctions are not allowed in such situations. For the data shown, this query returns the value 9.

Dealing with Empty Predicates

There is a subtle issue raised if the condition for the count function is never satisfied. You might expect that count would return a value of zero. In this case, however, the query is considered to have failed, and no value is assigned to the function result. For example, the following query has a condition that can never be satisfied, so the query result is simply empty, instead of being the number 0:

```
_[] = n <- agg<<n = count()>> Monarch(m), !Monarch(m).
```

Although this example is silly, there are situations where you would like to see a value of zero. As a simple example, consider Table 2.10.

TABLE 2.10 Descendants of Princess Diana and Prince Charles

Person	**Children**
Princess Diana	Prince William, Prince Harry
Prince Charles	Prince William, Prince Harry
Prince William	Prince George
Prince Harry	
Catherine, Duchess of Cambridge	Prince George

We may code this report as follows. (Strictly speaking, we should add the ring constraints on parenthood discussed previously, but for simplicity let's ignore these for now.)

```
// Schema
Person(p), hasPersonName(p:n) -> string(n).
isParentOf(p1, p2) -> Person(p1), Person(p2).

// Data
+isParentOf("Princess Diana", "Prince William").
+isParentOf("Princess Diana", "Prince Harry").
+isParentOf("Prince Charles", "Prince William").
+isParentOf("Prince Charles", "Prince Harry").
+isParentOf("Prince William", "Prince George").
+isParentOf("Catherine, Duchess of Cambridge",
    "Prince George").
```

Now suppose we want to use the count function to derive, for each person, how many children that person has. To begin, we might try adding the following code to the schema:

```
nrChildrenOf[p] = n -> Person(p), int(n).
nrChildrenOf[p] = n <- agg<<n = count()>>
   isParentOf(p, _).
// The number of children of person p =
// the count of parenthood facts for person p.
```

If you run the program with the data shown and then print the Person and nrChildrenOf predicates, you get the following results:

```
/— - start of Person facts— -\
   Prince Charles
   Prince Harry
   Prince William
   Princess Diana
   Prince George
   Catherine, Duchess of Cambridge
\— — end of Person facts— —

/— - start of nrChildrenOf facts— -\
   Prince Charles,                  2
   Princess Diana,                  2
   Prince William,                  1
   Catherine, Duchess of Cambridge, 1
\— — end of nrChildrenOf facts— — /
```

As expected, we see that Prince Charles and Princess Diana each have two children and that Prince William and "Princess Kate" have one. However, Prince Harry and Prince George are excluded from the nrChildrenOf output because they do not play the parent role in any parenthood facts. We might instead want to include them with a value of zero for their number of children. One way to do this is to split the nrChildrenOf rule into two pieces—one to handle the case of parents with children and another to deal with people who do not have children.

To implement the first part of this strategy, we rename the nrChildrenOf predicate to positiveNrChildrenOf to deal with the case where the number of children is greater than zero:

```
positiveNrChildrenOf[p] = n -> Person(p), int(n).
positiveNrChildrenOf[p] = n <- agg<<n = count()>>
    isParentOf(p, _).
// The positive number (i.e. number above 0) of
// children of person p = the count of parenthood
// facts for person p.
```

We can then derive the nrChildrenOf predicate using two rules, one for this case and one for the case of no children, as shown below:

```
nrChildrenOf[p] = n -> Person(p), int(n).
nrChildrenOf[p] = 0 <- Person(p), !isParentOf(p, _).
nrChildrenOf[p] = positiveNrChildrenOf[p] <-
    isParentOf(p, _).
// The number of children of p = 0
// if p is a person who is not a parent of someone,
// else it's the positive number of children of p.
```

With this change, a print of the nrChildrenOf predicate displays the intended results:

```
/— - start of nrChildrenOf facts— -\
   Princess Diana,                 2
   Prince William,                 1
   Prince Charles,                     2
   Catherine, Duchess of Cambridge, 1
   Prince Harry,                       0
   Prince George,                      0
\— — end of nrChildrenOf facts— — /
```

Be aware that there is a subtle performance cost to taking the above approach. For each `Person` entity, both of the `nrChildrenOf` rules must search the `isParentOf` predicate, even though we know that the `Person` cannot satisfy both rules (a person cannot both be a parent and not be a parent). In Appendix J, we will revisit this example to show how this extra cost can be avoided.

Computing Totals and Averages

This section discusses the `total` function, as well as how it can be used together with the `count` function to compute averages. The `total` function operates on a collection of facts and returns a single value; but unlike the `count` function, `total` requires an individual variable as a key argument. If *x* and *y* are individual variables and *Cx* denotes a condition in which *x* is a variable, then the following syntax is used to assign the value of `total(x)` to *y* when the condition *Cx* is true:

```
agg<<y = total(x)>> Cx // y = total(Cx) where Cx is true.
```

The `total` function sums numeric values, and therefore `total(x)` is legal only if *x* is a numeric variable. Table 2.11 lists the numeric datatypes supported by LogiQL.

For floating-point numbers that include an exponent, "`E`" precedes the power of 10. For example, the universal gravitational constant 6.67×10^{-11} may be written `6.67E-11.f`. Note that to distinguish literal values of types `float`, an explicit suffix `f` is required. Literals of type `decimal` may optionally include the suffix `d`.

TABLE 2.11 LogiQL Numeric Datatypes

Numeric Datatype	**LogiQL Syntax**	**Examples**
Integer	int	−3, 0, 35
Floating Point	float	0.0f, 31.555f, 2.9979E8f, 6.67E-11f
Decimal	decimal	0.0, 567.99d

TABLE 2.12 Expense Report

Item	Expense (US$)
Travel	300.50
Accommodation	300.50
Meals	100.20
Total	701.20

There are also built-in predicates for converting among types. The format for usage is that the names of the two types precede the name `convert`, as in the following example that converts a `string` literal to `decimal`:

```
+a[] = string:decimal:convert["52.43"].
```

As another example of the `total` function, consider the expense report presented in Table 2.12. Currently, LogiQL allows the `total` function only on numeric values and has no built-in support for units of measure, so we represent U.S. dollar amounts simply by numeric values. We use the `decimal` datatype to encode expense amounts. A basic schema and input data for the report is shown below:

```
// Schema
Item(i), hasItemName(i:n) -> string(n).
expenseOf[i] = e -> Item(i), decimal(e).
Item(i) -> expenseOf[i] = _.
// Each item has an expense.

// Data
+expenseOf["Travel"] = 300.50.
+expenseOf["Accommodation"] = 300.50.
+expenseOf["Meals"] = 100.20.
```

To derive the total expense, we add the following code to the schema, using the function `totalExpense[]` to store the derived total, and using the `total` function to sum over each expense amount:

```
totalExpense[] = t -> decimal(t).
totalExpense[] = t <- agg<<t = total(e)>>
    expenseOf[_] = e.
// The total expense is the sum of the item expenses.
```

TABLE 2.13 Multiple Expense Claims

Expense Claim	Item	Expense (US$)	Total (US$)
1	Travel	300.50	
	Accommodation	300.50	
	Meals	100.20	701.20
2	Travel	55.05	
	Meals	30.10	85.15

Note that 300.50 appears twice in the expense column of the above table. When the `total` function is applied to the asserted facts, it adds each of the two duplicate values just like any other value. So if you compile the program with the data shown and then print or query the `totalExpense` predicate, you get the correct value 701.2 (not 400.7) as the result.

Note also that LogiQL requires the result datatype for the `total` function to be the same as the input datatype of the component being summed. For example, changing the datatype of `totalexpense` in the above code to be `float` results in an error.

Now consider the report in Table 2.13 that deals with multiple expense claims, where we wish to total the expenses for each claim.

Using the ternary predicate `claimItemExpense[c,i]=e` to indicate that claim `c` for item `i` has expense `e` and the expression `totalExpenseOf[c]=t` to indicate that claim `c` has a total expense of `t`, we may code this report as follows:

```
Claim(c), hasClaimNr(c:n) -> int(n).
Item(i), hasItemName(i:n) -> string(n).
claimItemExpense[c, i] = e -> Claim(c), Item(i),
    decimal(e).
Claim(c) -> claimItemExpense[c, _] = _.
// Each claim has an item with an expense.
totalExpenseOf[c] = t -> Claim(c), decimal(t).
totalExpenseOf[c] = t <-
    agg<<t = total(e)>> claimItemExpense[c, _] = e.
// The total expense of claim c is the sum of
// the item expenses on claim c.
```

The input data for the report may be asserted as follows:

```
+claimItemExpense[1, "Travel"] = 300.50.
+claimItemExpense[1, "Accommodation"] = 300.50.
+claimItemExpense[1, "Meals"] = 100.20.
```

```
+claimItemExpense[2, "Travel"] = 55.05.
+claimItemExpense[2, "Meals"] = 30.10.
```

If you run the program with the data shown, and then print the `total ExpenseOf` predicate, you get the following:

```
/— - start of totalExpenseOf facts— -\
   2, 85.15
   1, 701.2
\— — end of totalExpenseOf facts— — /
```

Now consider Table 2.14 containing estimated IQs for some famous people (for more listings, see www.kids-iq-tests.com/famous-people.html). Suppose we wish to compute the average IQ of those listed. LogiQL does not include a function to compute averages, but we can use the `total` function to sum the IQs, and the `count` function to count the number of IQs, and then divide the sum by the count to compute the average IQ.

As a first attempt, we could try the following, using `iqOf[p] = iq` to mean the IQ of person `p` is `iq`. Assuming all IQs are positive and none exceeds 255, we treat IQ values as `ints` and add constraints on the maximum and minimum IQ:

```
// Schema
Person(p), hasPersonName(p:n) -> string(n).
iqOf[p] = iq -> Person(p), int(iq).
iqOf[_] = iq -> 0 < = iq, iq <= 255.
// Each IQ > = 0 and <= 255.
Person(p) -> iqOf[p] = _.      // Each person has an IQ.
nrPersons[] = n -> int(n).
nrPersons[] = n <- agg<<n = count()>> Person(_).
totalIQ[] = t -> int(t).
totalIQ[] = t <- agg<<t = total(iq)>> iqOf[_] = iq.
avgIQ[] = n -> int(n).
avgIQ[] = n <- n = totalIQ[]/nrPersons[].

// Data
+iqOf["Hillary Clinton"] = 140.
+iqOf["Albert Einstein"] = 160.
+iqOf["Bill Gates"] = 160.
```

If you run the program with the data shown and print or query the `avgIQ` predicate, you get 153, which omits the fractional part. This is because

TABLE 2.14 IQs of Famous People

Person	IQ
Hillary Clinton	140
Albert Einstein	160
Bill Gates	160

avgIQ is obtained by dividing one integer into another, and the *division* operator ('/') is then treated as integer division not floating-point division. If you wish to include the fractional part in the result of dividing one integer term x by another integer term y, you may instead use a conversion function (i.e., int:float:convert). For example, if you replace the derivation rule for avgIQ given above by the following line of code, declare avgIQ to be of type float, and then recompile the program and print or query the avgIQ predicate, you get 153.33333333333334, which includes the fractional part:

```
avgIQ[] = n <- n =
avgIQ[] = n <-
    n = int:float:convert[totalIQ[]]/
    int:float:convert[nrPersons[]].
```

Note that if the terms in the division are already typed as floating point (e.g., float instead of int), floating-point division is automatically performed when using "/" for the *division* operator. Also note that any attempt to divide by zero fails to return a result.

Tip: When using the count function, decide whether or not to return 0 if there are no facts to be counted.

Tip: When using the total function, ensure that the datatype of the numbers being summed is large enough to hold the final sum.

Tip: When using division, decide whether integer, decimal, or floating-point division is intended, and use the appropriate datatype.

Exercise 6A: Extend the program found in countMonarchs.logic with code to derive the predicate nrQueens[]=n to count the

TABLE 2.15 Products and Accessories

Product	Accessories
Paperback book	
Kindle	Cover, light, wireless

TABLE 2.16 Student Course Credits

Student Number	Course	Credit	Total Credit
101	CS100	5	
	MA100	5	
	PH101	4	14
202	CS100	5	
	JP100	4	9

number of queens. Then compile and run your program with the data in `genderOfData.logic` and print `nrQueens[]`. Include code to return 0 if there are no queens.

Exercise 6B: Write and test a program to record the facts in Table 2.15 and display the number of accessories (possibly 0) available for each product.

Exercise 6C: The report in Table 2.16 is used to maintain details about course credits obtained by students for the courses they have passed. A stub of a program for this report is shown here, as well as the relevant data. The stub can be found in `credits.logic`, and the data can be found in `creditsData.logic`. Extend the program to compute the total credit obtained by each student and print the results:

```
// Schema
Student(s), hasStudentNr(s:n) -> int(n).
Course(c), hasCourseCode(c:cc) -> string(cc).
hasCredit[c] = n -> Course(c), int(n).
passedCourse(s, c) -> Student(s), Course(c).

// Data
+hasCredit["CS100"] = 5, +hasCredit["MA100"] = 5.
+hasCredit["PH101"] = 4, +hasCredit["JP100"] = 4.
+passedCourse(101, "CS100"), +passedCourse(101,
    "MA100"), +passedCourse(101, "PH101").
+passedCourse(202, "CS100"), +passedCourse(202,
    "JP100").
```

Exercise 6D: The expense claim program and data discussed in this unit are stored as the files `expenses.logic` and `expensesData.logic`. Extend the program to compute the average expense of items on each of the two expense claims.

UNIT 2.7: CONSOLIDATION EXERCISE 2

This exercise gives you a chance to test how well you have mastered the topics covered in this chapter. The application domain is adapted from a previously published example by one of the authors.* The universe of discourse concerns a fragment of an information system used by a book publisher to maintain details about its books and personnel. Your task is to model this domain in LogiQL.

PART 1: BOOKS

One requirement of the information system is to produce reports about books that are either already published by the publisher or are under consideration for publication. Each of these books is identified by an International Standard Book Number (ISBN). For any given title, the publisher also has a policy of publishing at most one book with that title in any given year; for example, only one book published by the company in 2010 may be titled *Informatics*. It is possible that a book is directly translated from another book written in a different language. A sample extract of this report is shown in Table 2.17.

TABLE 2.17 Book Publication Details

ISBN	Title	Language	Published	Translated from
1-33456-012-3	*Mizu no Kokoro*	Japanese	2009	
2-55860-123-6	*Mind Like Water*	English	2010	1-33456-012-3
3-540-25432-2	*Informatics*	English	2010	
4-567-12345-3	*Semantics*	English	2011	
5-123-45678-5	*Informatics*	English	2011	
6-246-80246-8	*Geist wie Wasser*	German	2011	2-55860-123-6
7-345-10267-4	*Geest als Water*	Dutch		6-246-80246-8
8-543-21012-3	*Mens quasi Aquam*	Latin		2-55860-123-6

* T. Halpin. “Fact-Oriented Modeling: Past, Present and Future.” *Conceptual Modelling in Information Systems Engineering*, Springer, 2007, pp. 19–38.

The predicate declarations and some constraints for this report are shown below:

```
Book(b), hasISBN(b:n) -> string(n).
bookTitleOf[b] = t -> Book(b), string(t).
Language(lang), hasLanguageName(lang:ln) -> string(ln).
languageOf[b] = lang -> Book(b), Language(lang).
publicationYearOf[b] = yr -> Book(b), int(yr).
translationParentOf[b1] = b2 -> Book(b1), Book(b2).
Book(b) -> bookTitleOf[b] = _, languageOf[b] = _.
// Each book has a title and is written in a language.
```

The file and data for this program are accessible as `Q1.logic` and `Q1Data.logic`.

Q1: Extend the program with code to express the following external uniqueness constraint: For each book title and year, at most one book of that title is published in that year. Also, add code to capture the following exclusion constraint: If a book is directly translated from another book, its language must differ from that of the other book. The answers can be found in the file `Q1Answer.logic`.

Q2: Extend the program to derive the predicate `isaTranslationOf` (covering direct and indirect book translations) representing the transitive closure of `translationParentOf`. Constrain the direct translation predicate to be acyclic. Test your code by trying to add data that violates the constraint. Explain why there is no need to add an intransitivity constraint. The answers are in `Q2Answer.logic` and `Q2Counterexample.logic`.

Q3: Write a query to compute the number of books that are translated (directly or indirectly) from the book titled *Mizu no Kokoro* that was published in 2009. The answer is in `Q3Answer.logic`.

Q4: Write a query to list the ISBN and title of each book that is translated (directly or indirectly) from the book with ISBN 1-33456-012-3. The answer is in `Q4Answer.logic`.

PART 2: BOOK SALES

A report is also maintained on sales of published books. The extract from this report in Table 2.18 shows the sales figures for those published books in the previous report for which sales figures are available.

TABLE 2.18 Book Sales Information

ISBN	Year	Number Sold	Cumulative Sales	Best Seller?
1-33456-012-3	2009	5000		
	2010	6000		
	2011	5000	16000	Yes
2-55860-123-6	2010	3000		
	2011	4000	7000	No
3-540-25432-2	2010	1000		
	2011	2000	3000	No

The publisher considers any book that has sold at least 10,000 copies to be a best seller.

Q5: Write a program to model this report. Extend the file `Q5.logic` by introducing `PublishedBook` as a derived subtype of `Book`. Use the ternary predicate `yearlySales[b,yr]=n` to record the number of copies sold for a given book in a given year, ignoring books with no sales. Use `cumulativeSalesOf[b]=n` for computing the total number of copies sold to date for a given book with sales. Use the unary predicate `isaBestSeller` to derive whether a published book is a best seller. For testing purposes, you may use the associated data file `Q5Data.logic`. The answer is in the file `Q5Answer.logic`.

Q6: Add a rule to derive `nrSalesYearsOf[b]`, which computes for each sold book `b`, the positive number of years that book has sales figures. Then add a rule to derive `avgYearlySalesOf[b]`, which computes, for each book `b` with sales figures, the average number of copies sold per year. The answer is in `Q6Answer.logic`.

PART 3: PERSONNEL

Reports are also maintained on persons who perform work for the company as employees and/or authors and/or translators. The publisher identifies such persons by a personnel number (`PNr`). If a person has a Social Security Number (SSN), this is also recorded. Table 2.19 is an extract from a personnel report.

The predicate declarations and some constraints for this report are shown below. The `applicableGenderOf` predicate is used to indicate which person titles are restricted to which genders. For example, “Mr” is restricted to males and “Mrs” to females, while “Dr” has no such restriction.

TABLE 2.19 Personnel Information

PNr	Name	SSN	Title	Gender
1	John Smith	123-45-6789	Mr.	M
2	Don Bradchap	246-80-2468	Sir	M
3	Sue Yakamoto		Mrs.	F
4	Yoko Oya		Dr.	F
5	Isaac Seldon		Dr.	M
6	Fumie Mifune	678-90-1234	Dr.	F
7	John Smith	789-01-2345	Mr.	M
8	Ann Jones		Ms.	F
9	Selena Most		Mrs.	F
10	Julius Antony	100-01-2345	Mr.	M
11	Bernhard Schmidt		Dr.	M
12	Herman van Reit		Dr.	M

Note: M, male; F, female.

Though not required, we declared the `PersonTitle` predicate with a `string` refmode, instead of simply using `string` in its place, since this better explains the use of the `applicableGenderOf` predicate. However, doing this requires that we populate the `PersonTitle` predicate explicitly with data:

```
Person(p), hasPNr(p:n) -> int(n).
Gender(g), hasGenderCode(g:gc) -> string(gc).
PersonTitle(pt) -> string(pt).
personNameOf[p] = pn -> Person(p), string(pn).
ssnOf[p] = ssn -> Person(p), string(ssn).
personTitleOf[p] = pt -> Person(p), PersonTitle(pt).
genderOf[p] = g -> Person(p), Gender(g).
applicableGenderOf[pt] = g -> PersonTitle(pt),
    Gender(g).
Person(p) -> personNameOf[p] = _,
personTitleOf[p] = _, genderOf[p] = _.
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
```

The program and data are accessible as `Q7.logic` and `Q7Data.logic`:

Q7: Add constraint code to ensure that the `ssnOf` predicate is inverse-functional (i.e., each SSN applies to at most one person). Also add constraint code to enforce the following subset constraint: If a person has a person title that applies only to a specific gender, then that person must be of that gender. The answer is in `Q7Answer.logic`.

TABLE 2.20 Personnel Roles

ISBN	Author PNrs	Translator PNrs	Reviewer PNrs
1-33456-012-3	3		1, 4
2-55860-123-6		3	
3-540-25432-2	5		1, 7
4-567-12345-3	5, 8		7, 11
5-123-45678-5	7		5, 8
6-246-80246-8		11	
7-345-10267-4		12	
8-543-21012-3		10, 12	

The report in Table 2.20 indicates who plays what roles (author, translator, reviewer) for what books. If a book is a translation of another book, then its author(s) is/are not explicitly recorded.

There are many different ways to model this report, but let us choose to use the three predicate declarations shown below. The full program up to this point, including the previous code and these new predicate declarations, is accessible as the file `Q8.logic`. The associated data file is `Q8Data.logic`:

```
authored(p, b) -> Person(p), Book(b).
translatedFor(p, b) -> Person(p), Book(b).
reviewed(p, b) -> Person(p), Book(b).
```

Q8: Without introducing any more subtypes, add code to the above to enforce the following constraints: **Q8a:** the subset constraint that each book with a translator is translated from another book (use the predicate `translationParentOf`); **Q8b:** the subset constraint that each reviewed book has an author; **Q8c:** the *inclusive-or* constraint that each book is either authored or translated, ignoring the exclusive aspect; **Q8d:** the exclusion constraint that no person may author and review the same book. The answer is in `Q8Answer.logic`.

Q9: Write a query to list each book that is a translation of another book, as well as the PNr and name of the original author(s). *Hint:* Make use of the `isaTranslationOf` predicate. The answer is in `Q9Answer.logic`.

ANSWERS TO EXERCISES

Answer to Exercise 1A:

```
isFatherOf(p1, p2) -> Person(p1), Person(p2).
// If p1 is father of p2, then p1 and p2 are
// persons.

isFatherOf(p1, p), isFatherOf(p2, p) -> p1 = p2.
// Each person has at most one father.
```

Answer to Exercise 1B:

```
fatherOf[p1] = p2 -> Person(p1), Person(p2).
// If the father of p1 is p2, then p1 and p2 are
// persons. Each person has at most one father.

isFatherOf(p1, p2) <- fatherOf[p2] = p1.
// p1 is father of p2 if the father of p2 is p1.
```

Answer to Exercise 2A:

```
inPositionHasGivenName[_, n] = _ -> n > 0, n < 10.
// If something in position n has some given name
// then the integer n is at least 1 and at most 9.
```

Note: The constraint may also be written as follows:

```
inPositionHasGivenName[_, n] = _ -> n > = 1,
    n < = 9.
```

Answer to Exercise 2B:

```
inPositionHasGivenName[m, n1] = gn,
inPositionHasGivenName[m, n2] = gn -> n1 = n2.
// Each given name of a monarch appears in only one
// position in that monarch's list of given names.
```

Note the occurrence of multiple atoms on the left-hand side of this constraint.

Answer to Exercise 2C:

```
sharesGivenName(m1, m2) -> Monarch(m1), Monarch(m2).
sharesGivenName(m1, m2) <-
  hasGivenName(m1, n), hasGivenName(m2, n),
  m1 ! = m2.
// Monarchs m1 and m2 share a given name if they are
// different monarchs and there is a given name n
// that both of them have.
```

Answer to Exercise 2D:

```
sharesGivenName2(m1, m2) -> Monarch(m1), Monarch(m2).
sharesGivenName2(m1, m2) <-
    hasGivenName(m1, n), hasGivenName(m2, n),
    hasMonarchName(m1:n1), hasMonarchName(m2:n2),
    n1 < n2.
// Monarchs m1 and m2 share a given name if there is a
// given name n that both of them have and if the
// first Monarch name is less than the second's.
```

Answer to Exercise 3A:

```
Employee(e), hasEmpNr(e:en) -> string(en).
ssnOf[e] = ssn -> Employee(e), string(ssn).
passportNrOf[e] = ppn -> Employee(e), string(ppn).

ssnOf[e1] = ssn, ssnOf[e2] = ssn -> e1 = e2.
// Each Social Security Number is for only one
// employee.

passportNrOf[e1] = ppn, passportNrOf[e2] = ppn ->
    e1 = e2.
// Each passport nr is for only one employee.

Employee(e) -> ssnOf[e] = _ ; passportNrOf[e] = _.
// Each employee has a Social Security Number
// or a passport number.
```

Answer to Exercise 3B:

```
// Schema
Store(s), hasNeighborhood(s:name) -> string(name).
```

```
xCoordinateOf[s] = x -> Store(s), int(x).
yCoordinateOf[s] = y -> Store(s), int(y).

// Each (x, y) coordinate pair applies to only one
// store.
xCoordinateOf[s1] = x, yCoordinateOf[s1] =
    y, xCoordinateOf[s2] = x,
    yCoordinateOf[s2] = y -> s1 = s2.

// Data
+xCoordinateOf["A"] = 2, +yCoordinateOf["A"] = 3.
+xCoordinateOf["B"] = 1, +yCoordinateOf["B"] = 1.
+xCoordinateOf["C"] = 4, +yCoordinateOf["C"] = 1.

// Query
_(s, x, y) <- xCoordinateOf[s] = x,
    yCoordinateOf[s] = y.
```

Result:

```
C, 4, 1 B, 1, 1 A, 2, 3
```

Answer to Exercise 4A:

```
Company(c), hasCompanyName(c:n) -> string(n).
Product(p), hasProductName(p:n) -> string(n).
carries(c, p) -> Company(c), Product(p).
Dealer(c) -> Company(c).
Dealer(c) <- carries(c, _).
NonDealer(c) -> Company(c).
NonDealer(c) <- Company(c), !Dealer(c).
// Instead of !Dealer(c) you may use !carries(c, _).
IPhoneDealer(c) -> Dealer(c).
IPhoneDealer(c) <- carries(c, "IPhone").
```

Results:

```
Company:        Sears, Target, KMart
Dealer:         Sears, Target
NonDealer:      KMart
IPhoneDealer:   Sears
```

Answer to Exercise 4B:

```
provides(s, c) -> owns(s, c).
```

Answer to Exercise 5A:

```
_(p) <- isAncestorOf(p, "Prince William").
```

Results:

```
Prince Charles
Princess Diana
John Spencer
Frances Kydd
Elizabeth II
Prince Phillip
Edmond Roche
Ruth Roche
Albert Spencer
Cynthia Spencer
Prince Andrew
Princess Alice
George VI
Elizabeth Bowes-Lyon
William Gill
Ruth Gill
James Roche
Frances Work
James Hamilton
Rosalind Hamilton
Charles Spencer
Margaret Baring
Prince Louis
Princess Victoria
George I of Greece
Olga Constantinovna
Claude Bowes-Lyon
Cecilia Bowes-Lyon
George V
Mary of Teck
```

Answer to Exercise 5B:

```
// Schema
Product(p), hasProductCode(p:pc) -> string(pc).
directlyContainsIn[p1, p2] = qty -> Product(p1),
    Product(p2), int(qty).
```

```
contains(p1, p2) <- directlyContainsIn[p1, p2] = _.
contains(p1, p2) <- directlyContainsIn[p1, p3] = _,
    contains(p3, p2).
    !contains(p, p).

// Query
_(p, q) <- contains(p, q).
```

Results:

```
B, C
A, C
C, F
B, F
A, F
C, E
B, E
A, E
A, B
B, D
A, D
```

Answer to Exercise 5C:

```
isParentOf(p1, p2), isAncestorOf(p2, p3) ->
    !isParentOf(p1, p3).
```

Answer to Exercise 6A:

```
nrQueens[] = n -> int(n).
nrQueens[] = n <- agg<<n = count()>> Monarch(m),
    genderOf[m] = "F".
// nrQueens = count of female monarchs.
```

Result: 3

Answer to Exercise 6B:

```
positiveNrAccessoriesOf[p] = n -> Product(p), int(n).
positiveNrAccessoriesOf[p] = n <-
agg<<n = count()>> features(p, _).
nrAccessoriesOf[p] = 0 <- Product(p), !features(p, _ ).
```

```
  nrAccessoriesOf[p] = positiveNrAccessoriesOf[p] <-
      features(p, _).
```

Answer to Exercise 6C:

```
  Student(s), hasStudentNr(s:n) -> int(n).
  Course(c), hasCourseCode(c:cc) -> string(cc).
  hasCredit[c] = n -> Course(c), int(n).
  passedCourse(s, c) -> Student(s), Course(c).
  totalCreditOf[s] = t -> Student(s), int(t).
  totalCreditOf[s] = t <-
      agg<<t = total(n)>> passedCourse(s, c),
      hasCredit[c] = n.
  // The total credit of student s is the sum of
  // the credit points of the courses passed by s.
```

Result:

```
  /— - start of totalCreditOf facts— -\
  202, 9
  101, 14
  \— — end of totalCreditOf facts— — /
```

Answer to Exercise 6D:

```
  Claim(c), hasClaimNr(c:n) -> int(n).
  Item(i), hasItemName(i:n) -> string(n).
  claimItemExpense[c, i] = e -> Claim(c), Item(i),
      decimal(e).
  Claim(c) -> claimItemExpense[c, _] = _.
  // Each claim has an item with an expense.
  totalExpenseOf[c] = t -> Claim(c), decimal(t).
  totalExpenseOf[c] = t <-
      agg<<t = total(e)>> claimItemExpense[c, _] = e.
  // The total expense of claim c is the sum of
  // the item expenses on claim c.
  nrItemsOf[c] = n -> Claim(c), int(n).
  nrItemsOf[c] = n <-
      agg<<n = count()>> claimItemExpense[c, _] = _.
  avgExpenseOf[c] = a -> Claim(c), decimal(a).
  avgExpenseOf[c] = avgExp <-
      avgExp = totalExpenseOf[c]/nrItemsOf[c].
```

Result:

```
/— - start of avgExpenseOf facts— -\
 2, 42.575
 1, 233.733
\— — end of avgExpenseOf facts— — /
```