CHAPTER 3

Diving Deeper

CONTENTS

Unit 3.1: The `max` and `min` Functions 84
- Argmin and Argmax 87
- Summary of Aggregation Functions 88

Unit 3.2: Safety Conditions for Rules and Queries 89
Unit 3.3: Derivation Rule Semantics 95
Unit 3.4: Delta Rules and Pulse Predicates 102
- Delta Modifiers 103
- Delta Logic 105
- Pulse Predicates 105

Unit 3.5: Transaction Processing 109
- Transactions 110
- Transaction Processing Example 113
- Stage Suffixes 117
- Summary 118

Unit 3.6: Additional Built-in Operators and Functions 120
- Arithmetic 120
- String Manipulation 121
- Aggregation Functions for Ordering 124

Unit 3.7: Consolidation Exercise 3 130
Answers to Exercises 134
- Answer to Exercise 1A 134
- Answer to Exercise 1B 134
- Answer to Exercise 2A 135
- Answer to Exercise 2B 135

Answer to Exercise 3 135
Answer to Exercise 4A 136
Answer to Exercise 4B 136
Answer to Exercise 4C 137
Answer to Exercise 5A 137
Answer to Exercise 5B 137
Answer to Exercise 5C 138
Answer to Exercise 5D 138
Answer to Exercise 6A 139
Answer to Exercise 6B 139
Answer to Exercise 6C 139

This chapter builds on the concepts and syntax of the LogiQL language considered in the previous chapters, introducing some more advanced features of the language and describing how programs are executed by the LogiQL engine. The first unit discusses aggregation functions for computing minima and maxima. We then identify some safety conditions to ensure that rules and queries execute in a finite time. After this, a description is given of how derivation rules are processed. Then come two units relating to the handling of changes to the EDB. The first examines how delta rules and pulse predicates may be used to make changes to the database, and the second provides a simple explanation of how transactions are supported in LogiQL. The final unit considers some further built-in operators and functions (scalar or aggregation) that can be useful. The consolidation exercise at the end gives you an opportunity to test your mastering of the new concepts and syntax considered in the chapter.

UNIT 3.1: THE `max` AND `min` FUNCTIONS

This unit discusses two additional aggregation functions, `max` and `min`, that are used to find the minimum or maximum value that satisfies a specified condition. They are applied using the same syntax that we have seen previously: If x and y are individual variables, f denotes one of the `max` or `min` functions, and `Cx` denotes a condition in which `x` is used as a variable, then the following syntax is used to assign the value of `f(x)` to `y` when the condition `Cx` is true:

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

TABLE 3.1 IQs of Famous People

Person	IQ
Hillary Clinton	140
Albert Einstein	160
Bill Gates	160

As a simple example, consider Table 3.1 of proposed IQs, which we coded earlier using the following predicate declarations:

```
Person(p), hasPersonName(p:n) -> string(n).
iqOf[p] = iq -> Person(p), int(iq).
```

Using `maxIQ[]` to denote the maximum IQ and `minIQ[]` for the minimum IQ, we can use the `max` and `min` functions in aggregation rules to derive these values as follows:

```
maxIQ[] = n -> int(n).
maxIQ[] = n <- agg<<n = max(iq)>> iqOf[_] = iq.
// maxIQ equals the maximum IQ of any person.

minIQ[] = n -> int(n).
minIQ[] = n <- agg<<n = min(iq)>> iqOf[_] = iq.
// minIQ equals the minimum IQ of any person.
```

If you run the program with the data shown and then print or query the `maxIQ` and `minIQ` predicates, you will get 160 and 140, respectively.

Unlike the `total` function, which applies only to numeric values, the `max` and `min` functions may be used with values of any ordered datatype (e.g., where the *less than* operator (`'<'`) is defined). For example, character strings may be ordered alphabetically using <, so we may derive the maximum (sorted last, alphabetically) person name as follows:

```
maxPersonName[] = n -> string(n).
maxPersonName[] = n <- agg<<n = max(pn)>>
    hasPersonName(_:pn).
// maxPersonName is the person name that is last
// when person names are ordered alphabetically.
```

For the data shown, querying `maxPersonName` returns `"Hillary Clinton"`.

Now that maxIQ[] has been derived, we may use it to derive who are the brightest people (in the sense of having the highest IQ), as follows:

```
isBrightest(p) -> Person(p).
isBrightest(p) <- iqOf[p] = maxIQ[].
// Person p is brightest if the IQ of p is the maximum
// IQ.
```

For the data shown, printing the isBrightest predicate returns the following:

```
/— - start of isBrightest facts— -\
   Bill Gates
   Albert Einstein
\— — end of isBrightest facts— — /
```

Now consider Table 3.2, which is an extended report on IQs based on the same Web site used earlier, which adds gender details.

To include the gender facts, we extend the previous program with the following code:

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

The following aggregation rule may now be used to derive the maximum IQ for each gender:

```
maxIQof[g] = n -> Gender(g), int(n).
maxIQof[g] = n <- agg<<n = max(iq)>> iqOf[p] = iq,
    genderOf[p] = g.
// maxIQof gender g is maximum IQ of people of
// gender g.
```

TABLE 3.2 Genders and IQs of Famous People

Person	Gender	IQ
Albert Einstein	M	160
Bill Gates	M	160
Dwight Eisenhower	M	122
Marilyn vos Savant	F	228
Hillary Clinton	F	140
Nicole Kidman	F	132

Note: M, male; F, female.

Note that the condition for an aggregation function may be a conjunction, as in the example above. When run with the data in the extended report, a print of the `maxIQof` predicate returns the following results:

```
/— - start of maxIQof facts— -\
  F, 228
  M, 160
\— — end of maxIQof facts— — /
```

The code for the complete IQ program discussed in this unit is available as `IQ3.logic` and the data is available as `IQ3data.logic`.

Argmin and Argmax

Suppose that in the above IQ example `maxIQ` is not yet defined and we want to compute not only the maximum IQ but also the person who has it. Would it be acceptable to include this computation in the following rule?

```
isBrightest(p) -> Person(p).
isBrightest(p) <- iqOf[p] = n, agg<<n = max(iq)>>
    iqOf[_] = iq.
```

No! The above code generates an error because of the following syntax requirement. If an aggregation function definition is used in the body of a rule, it must comprise the whole body of the rule. The aggregation function definition (of the form `agg<<y = f(x)>> Cx`) cannot be combined with any other formula in the body. This requirement is violated in the above rule because another conjunct (`iqOf[p]=n`) is included in the rule body.

When computing a maximum value, it is often desirable to determine not only the value but also the item that has that value. For example, in computing the maximum IQ value, we also wanted to know who has that value. The set of one or more instances of a function argument that returns the maximum value for that function is said to be the *argmax* of the function. For example, the set of person(s) with the highest IQ is the *argmax* of the `iqOf` function. Similarly, the set of one or more instances of a function argument that returns the minimum value for that function is the *argmin* of the function. Background on the use of argmax and argmin in mathematics may be found in the Wikipedia article on "Arg_max."

Intuitively, for a given function, its argmax or argmin may be computed in two steps, by first computing the maximum or minimum value,

respectively, and then determining the set of item(s) associated with that maximum/minimum value. For example, above we used the `max` function to compute `maxIQ`, the maximum value returned from `iqOf`, and then used the `isBrightest` rule to derive the person(s) with that maximum IQ.

Summary of Aggregation Functions

The aggregation functions considered so far are summarized in Table 3.3.

Tip: When including aggregation function definitions in the body of a rule, ensure that nothing else is included in the rule body.

Exercise 1A: A program and data for the following expense report are shown in Table 3.4. You may also access them as the files `expenses.logic` and `expensesData.logic`. Extend the program to derive which item(s) is/are the most expensive, run the program with the data shown, and print the result:

```
// 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.
```

TABLE 3.3 Aggregation Functions

Aggregation Function	Result Returned
`count()`	Number of instances satisfying the specified condition
`total(x)`	Sum of numeric x values satisfying the specified condition
`max(x)`	Maximum x value satisfying the specified condition
`min(x)`	Minimum x value satisfying the specified condition

TABLE 3.4 Travel Expenses

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

TABLE 3.5 Expense Report Claims

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

Exercise 1B: The program and data for the expense claims for the report in Table 3.5 are available as the files `expenses5.logic` and `expenses5Data.logic`. Extend the program to compute, for each claim, the most expensive item(s) for that claim. Then run the program with the data supplied and print the results.

UNIT 3.2: SAFETY CONDITIONS FOR RULES AND QUERIES

As a programmer, you want to write high-quality code. This means not only that the code is correct, efficient, and well documented, but that it is robust (able to deal with a variety of inputs) and that it does not go into infinite loops. LogiQL constraints help you write robust programs, and the LogiQL compiler helps you ensure that your program does not run indefinitely. Roughly speaking, a derivation rule or query is **safe** if and only if it is guaranteed to return a finite result using a procedure that terminates in a finite time. In this unit we discuss a number of restrictions that the LogiQL compiler places on rules and queries to ensure that they are safe. If you try to compile a rule or query that violates any of these restrictions, you will get an error message.

This unit is primarily concerned with helping you avoid situations where a computation might be required to look at an infinite number of possibilities. If we have a simple predicate populated with a finite number of asserted facts, and we try to compute a subset of the facts satisfying a simple condition, then we do not run into any trouble. For example, if we have a list of people along with their IQ data and we ask for the IQ of a particular person, then we cannot get in trouble, even if the person is not in the database. The computation merely checks each of the finite number of asserted facts to produce its answer.

There are some features of LogiQL, however, that could potentially get you into trouble. This unit talks about several of them including negation, disjunction, and built-in datatypes. We begin by reviewing some of LogiQL's syntax.

Recall that an atom is the application of a predicate to a list of terms, such as variables (possibly anonymous) or literals. For example, the following are atoms:

```
Person(p)                    // p is a person.
loves(p, _)                  // p loves something.
loves(p, "Juliet")           // p loves Juliet.
loves("Romeo", "Juliet")     // Romeo loves Juliet.
```

An atom that is not in the scope of a *logical not* operator (`'!'`) is called a *positive atom*. An atom in the scope of a *logical not* operator is called a *negated atom*. Here are some examples of negated atoms:

```
!Person(x)                   // x is not a person.
!likes(p1, p2)               // p1 does not like p2.
!likes("Leopold Kronecker", "Georg Cantor")
// Leopold Kronecker does not like Georg Cantor.
```

Let us use the term **domain predicate** for a predicate specific to the application domain that can be populated with only a finite set of data. The facts that populate domain predicates are either explicitly asserted by the user (and hence their number is finite), or they are derived from other domain predicates using safe rules. Domain predicates are distinct from the predicates used for built-in datatypes (e.g., `int,string`) and predefined operations (e.g., `'=', '+', '<'`).

Let's look at a typical rule that takes the form of a head consisting of a positive atom and the body comprising a conjunction of one or more positive or negative atoms. For example, the body of the following rule is a conjunction of two positive atoms involving a domain predicate (`isParentOf`), and one negative atom involving a built-in equality predicate (`'='`).

```
isBrotherOf(x, y) -> Person(x), Person(y).
isBrotherOf(x, y) <- isParentOf(z, x),
    isParentOf(z, y), x ! = y.
// x is a brother of y if some z is a parent of x and y
// and x is not equal to y.
```

The following safety condition provides one syntactic check to help determine whether a rule or query of this type is safe.

SC1: *Each variable appearing in the head of a rule must also appear in the rule's body. Moreover, its appearance in the body must be in a positive context, as an argument of either a domain predicate or a domain equality. In the case of a domain equality, the other operand of the equality must be either a constant expression or a domain variable that occurs in a positive context.*

The purpose of this rule is to prevent situations where a variable in the body of a rule can take on an unlimited number of values. For example, the `isBrotherOf` derivation rule above satisfies this safety condition because each of its head variables (`x` and `y`) can only take values that arise from the `isParentOf` predicate. (Note that in the next chapter we discuss two cases, derived-only predicates and head existentials, where the above condition may be safely broken.)

As a simple example of a rule that violates the above safety condition, consider the following unsafe rule:

```
isFemale(x) <- !isMale(x).    // Unsafe rule!
// x is female if x is not male.
```

This illegal rule declares that everything that is not male is female. This is far too strong a rule, as it includes anything in the application domain that is not male (e.g., houses, names, etc.). Moreover, in an infinite domain, the result could be infinite. To fix the derivation rule, we need to restrict the variable to range over the relevant domain predicate (e.g., `Person` or `Animal`). For example, the following rule satisfies SC1 because its sole head variable (`x`) occurs in the body as the argument of the positive domain atom `Person(x)`:

```
isFemale(x) <- Person(x), !isMale(x).
// x is female if x is a person and not male.
```

Each of the following rules also violates SC1. Can you see why?

```
isText(x) <- string(x). // Unsafe rule!
positiveNumber(x) <- decimal(x), x > 1.0.
// Unsafe rule!
```

Although the head variable `x` appears in positive context in the body of the above rules, it appears only with built-in predicates, not domain predicates.

Conceptually, there are infinitely many strings and numbers, so the result is potentially infinite. Even though computer systems store strings and numbers using only a finite number of bits, for all practical purposes it would still be undesirable to deal with such a large number of possible strings or numbers.

The following two safe rules illustrate cases where a head variable does not appear in the body in a domain predicate, but it does appear in a simple equality, where the other operand is either a constant expression or a domain variable in a positive context:

```
isEven(n) <- n = 2.
uses(p, c) <- drives(p, b), b = c.
```

In the first rule, `n` can only take one value, the integer 2. In the second rule, `p` appears positively inside of the `drives` predicate. Although `c` is not inside of `drives`, it must be equal to `b`, and `b` is inside of `drives`. So `c` is similarly finitely bounded.

Now consider the following schema and data:

```
// Schema
Person(p), hasPersonName(p:n) -> string(n).
CarModel(cm), hasModelName(cm:n) -> string(n).
drives(p, cm) -> Person(p), CarModel(cm).
NonDriver(p) -> Person(p).
NonDriver(p) <- Person(p), !drives(p, _).
// A nondriver is a person who does not drive any car
// model.
lang:isEntity['NonDriver] = true.

// Data
+Person("Terry"), +Person("Norma"), +Person("Lee").
+drives("Terry", "Mazda").
+drives("Lee", "Ford").
```

If you run the program with the data shown, and then print the `NonDriver` predicate, you will obtain the correct result: `Norma`. The rule for `NonDriver` satisfies SC1 because the sole head variable (`p`) occurs in the body in the positive domain atom `Person(p)`. Note that an anonymous variable (_) occurs only in a negated atom, but this does not violate the safety condition because it is not a head variable.

Now consider the following two lines from our program. On first look, the use of the anonymous variable in the `NonDriver` rule indicates that a nondriver is a person who does not drive anything (not just car models). However, given the type declaration for the `drives` predicate, the LogiQL compiler can infer that anything driven is a car model, so there is no need to state that explicitly:

```
drives(p, cm) -> Person(p), CarModel(cm).
NonDriver(p) <- Person(p), !drives(p, _).
```

LogiQL is able to handle some variants of the above that involve more complex syntax. For example, it can handle simple negated conjunctions such as the following:

```
isUnsuitable(p) <- Person(p), !(isHardworking(p),
    isIntelligent(p)).
```

SC2: *Each named variable appearing in the scope of a negation within the body of a rule must also appear in a positive context in that rule body.*

In addition to conjunctions and simple negations, LogiQL allows the body of a rule to include logical disjunctions, whose main operator is *inclusive-or*. The language treats a disjunctive rule as shorthand for a set of simpler rules, each containing one of the disjuncts. In order for the original disjunctive rule to be safe, each of the simpler rules must be safe. Since each simple rule in the resulting rule set has the same head, the first safety condition above implies the following safety condition:

SC3: *Each disjunct in the body of a disjunctive derivation rule must include the same selection of head variables; the only other variables allowed are anonymous variables.*

The following safe disjunctive derivation rule satisfies this restriction:

```
isParentOf(p1, p2) <- fatherOf[p2] = p1 ;
    motherOf[p2] = p1.
```

This disjunctive rule is shorthand for the following two rules, both of which are safe:

```
isParentOf(p1, p2) <- fatherOf[p2] = p1.
isParentOf(p1, p2) <- motherOf[p2] = p1.
```

The following disjunctive rule, however, violates the above restriction:

```
p(x, y) <- q(x) ; r(y). // Unsafe!
```

Notice that if you try to rewrite this rule as a pair of simpler, non-disjunctive, rules each of the resulting rules violates our original safety condition:

```
p(x, y) <- q(x). // Unsafe!
p(x, y) <- r(y). // Unsafe!
```

As another variant, consider queries. A query is essentially a derivation rule where we do not care about the name of the head predicate, because we simply want to see the result of the derivation. So the safety restrictions discussed for derivation rules also apply to queries. For example, the following query to return male persons is safe, but the query after it is unsafe:

```
_(x) <- Person(x), !isFemale(x).   // Safe query
_(x) <- !isFemale(x).              // Unsafe query!
```

There is one other, subtler, situation you need to be aware of when constructing safe and robust programs. This situation arises when you have a collection of safe rules, which, when treated as a whole, can lead to a program that never terminates.

To understand this situation, consider a set of rules. A typical rule in the set has a right-hand side that refers to other predicates. If any of these other predicates is also derived, then the computation of the original rule depends on the computation of this derivation rule. This dependency may extend to predicates mentioned in the bodies of the new rules. A problem arises if the rules are mutually recursive, that is, if the overall set of dependencies is cyclic. In particular, if there is a cycle in the dependency structure that includes either a negated atom, or an aggregation, then the LogiQL compiler cannot guarantee termination. Hence, the following safety condition applies.

SC4: *Any cyclic dependencies in a LogiQL program should not contain negated atoms or aggregations.*

Here is a program sketch illustrating a violation of this safety condition:

```
Youth(y), hasYouthName(y:n) -> string(n).
Boy(b) -> Youth(b).
Girl(g) -> Youth(g).
Boy(x) <- Youth(x), !Girl(x).
Girl(x) <- Youth(x), !Boy(x).
```

Note that Boy depends upon Girl, that Girl depends upon Boy, and that negation is involved.

Imagine now the following fact assertions:

```
+Youth("Adam").
+Girl("Eve").
+Boy("Colin").
```

When the execution engine tries to compute a gender for Adam, a problem arises. Adam is a Youth that is not (initially) a Girl, which makes Adam a Boy. However, Adam is also not (initially) a Boy, which makes Adam a Girl, violating the premise of the computation that made Adam a Boy in the first place. To avoid situations like this from causing trouble, the LogiQL compiler issues an error message. Aggregations can lead to similar problems.

Tip: When declaring rules or queries, make sure they satisfy the safety conditions discussed in this unit.

Exercise 2A: Explain what is wrong with the following program, and modify it to fix the error:

```
Light(l), hasLightNr(l:n) -> int(n).
isOn(l) -> Light(l).
isOff(l) <- !isOn(l).
```

Exercise 2B: Identify the safety violations in each of the following rules:

```
p(x, y, z) <- q(x), !r(y), z > 1.
p(x, y, z) <- q(x), r(y), s(z); q(y), r(z).
```

UNIT 3.3: DERIVATION RULE SEMANTICS

This unit provides a simplified account of how the LogiQL engine uses derivation rules to compute new facts. The intent is to give you an idea of how your program is actually executed by the LogiQL execution engine. The example that we will use is a simple ancestry graph. Because the ancestry relationship is recursive, multiple computation steps are required to complete the computation.

In the graph shown in Figure 3.1, each node denotes a person, which we identify by a single given name. An arrow from a node to a node below it

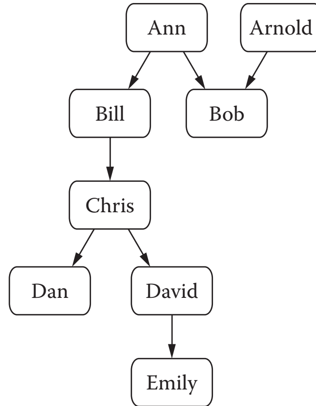

FIGURE 3.1 Parenthood example.

denotes a parenthood relationship (e.g., Ann is a parent of Bill and Bob). As our knowledge of parenthood is incomplete, some parenthood facts are missing (e.g., for most nodes, only one parent is shown).

The following code (available in `ancestry.logic` and `ancestryData.logic`) may be used to store these parenthood facts and derive who is an ancestor of whom. For simplicity, some ring constraints (e.g., nobody can be an ancestor of himself/herself) are omitted:

```
Person(p), hasPersonName(p:n) -> string(n).
isParentOf(p1, p2) -> Person(p1), Person(p2).
!isParentOf(p, p). // Nobody is his/her own parent.

isAncestorOf(p1, p2) <-
    isParentOf(p1, p2) ;
    isParentOf(p1, p3), isAncestorOf(p3, p2).
// p1 is an ancestor of p2 if p1 is a parent of p2 or
// p1 is a parent of some p3 who is an ancestor of p2.
```

Here are the corresponding fact assertions:

```
+isParentOf("Ann", "Bill").
+isParentOf("Ann", "Bob").
+isParentOf("Arnold", "Bob").
+isParentOf("Bill", "Chris").
+isParentOf("Chris", "Dan").
+isParentOf("Chris", "David").
+isParentOf("David", "Emily").
```

Visually, the descendants of a person node in the graph are the nodes on the path(s) starting at that person and proceeding downward. For example, the descendants of Bill are Chris, Dan, David, and Emily. Here is what you would see upon querying `isParentOf` and `isAncestorOf`.

```
isParentOf:                isAncestorOf:
   Chris, David               Chris, David
   David, Emily               Bill, David
   Bill, Chris                Ann, David
   Chris, Dan                 David, Emily
   Ann, Bill                  Chris, Emily
   Arnold, Bob                Bill, Emily
   Ann, Bob                   Ann, Emily
                              Bill, Chris
                              Ann, Chris
                              Chris, Dan
                              Bill, Dan
                              Ann, Dan
                              Ann, Bill
                              Arnold, Bob
                              Ann, Bob
```

From the seven parenthood facts, 15 ancestry facts were derived using the recursive rule for `isAncestorOf`. But how was this computation performed by the LogiQL execution engine? There are several possibilities, and the actual approach used is quite complex due to optimizations that the engine performs. We will now describe a simple bottom-up approach that approximates what the engine does. The approach iteratively applies the recursive rule to find new facts that are immediate consequences of the currently known facts until we reach a state called the **fixedpoint**, after which reapplication of the rule generates no new facts.

For convenience, the recursive rule is restated below. It can be thought of as two rules, one for each disjunct in the body, with the first rule providing the basis clause, and the second rule providing the recursive clause:

```
isAncestorOf(p1, p2) <-
    isParentOf(p1, p2) ;
    isParentOf(p1, p3), isAncestorOf(p3, p2).
// p1 is an ancestor of p2 if p1 is a parent of p2
// or p1 is a parent of some p3 who is an ancestor
// of p2.
```

Note that in the above, `isParentOf` is an EDB predicate, and `isAncestorOf` is an IDB predicate. (Its facts are computed by the above rule.)

When the `isAncestorOf` rule is first executed, the `isParentOf` predicate is populated with seven facts, and the `isAncestorOf` predicate is empty, which we picture as shown below:

```
isParentOf:                 isAncestorOf:
   Ann, Bill
   Ann, Bob
   Arnold, Bob
   Bill, Chris
   Chris, Dan
   Chris, David
   David, Emily
```

Applying the basis clause, `isAncestorOf(p1,p2)<-isParentOf (p1,p2)` instantiates `isAncestorOf` with the seven tuples in `isParentOf`. Applying the recursive clause has no effect in this iteration because `isAncestorOf` was empty when the rule began executing. So at the end of this first iteration, the database is as shown below. The new facts added in the iteration are highlighted in *italics*:

```
isParentOf:                   isAncestorOf:
   Ann, Bill                  Ann, Bill
   Ann, Bob                   Ann, Bob
   Arnold, Bob                Arnold, Bob
   Bill, Chris                Bill, Chris
   Chris, Dan                 Chris, Dan
   Chris, David               Chris, David
   David, Emily               David, Emily
```

The state of the database at the end of the first iteration is the same as the state at the start of the second iteration of the rule. From here on, the basis clause can add no new facts because we already have all the parenthood tuples in `isAncestorOf`, so we need consider only the recursive clause, that is,

```
isAncestorOf(p1, p2) <- isParentOf(p1, p3),
    isAncestorOf(p3, p2).
```

We can apply this rule to the database by looking for situations where a child in the second role of the `isParentOf` predicate matches an ancestor in the

first role of the `isAncestorOf` predicate. For example, Bill, child of Ann, is an ancestor of Chris, which makes Ann also an ancestor of Chris. These matches are depicted with matching numbers in the following display:

```
isParentOf:                 isAncestorOf:
   Ann, Bill (1)               Ann, Bill
   Ann, Bob                    Ann, Bob
   Arnold, Bob                 Arnold, Bob
   Bill, Chris (2)         (1) Bill, Chris
   Chris, Dan              (2) Chris, Dan
   Chris, David (3)        (2) Chris, David
   David, Emily            (3) David, Emily
```

As a result, the state of the database immediately after this second iteration is as shown below (just the facts added in this second iteration are highlighted):

```
isParentOf:                 isAncestorOf:
   Ann, Bill                   Ann, Bill
   Ann, Bob                    Ann, Bob
   Arnold, Bob                 Arnold, Bob
   Bill, Chris                 Bill, Chris
   Chris, Dan                  Chris, Dan
   Chris, David                Chris, David
   David, Emily                David, Emily
                               Ann, Chris
                               Bill, Dan
                               Bill, David
                               Chris, Emily
```

In the third iteration, applying the recursive clause to the database results in three new ancestry facts being derived by matching a child in a parenthood fact with an ancestor in one of the new ancestry facts, as shown below by the matching numbers:

```
isParentOf:                 isAncestorOf:
   Ann, Bill (1)               Ann, Bill
   Ann, Bob                    Ann, Bob
   Arnold, Bob                 Arnold, Bob
   Bill, Chris (2)             Bill, Chris
   Chris, Dan                  Chris, Dan
   Chris, David                Chris, David
   David, Emily                David, Emily
                               Ann, Chris
```

```
                                   (1) Bill, Dan
                                   (1) Bill, David
                                   (2) Chris, Emily
```

As a result, the state of the database immediately after this third iteration is as shown below (just the new facts added in this third iteration are highlighted):

```
isParentOf:                 isAncestorOf:
   Ann, Bill                   Ann, Bill
   Ann, Bob                    Ann, Bob
   Arnold, Bob                 Arnold, Bob
   Bill, Chris                 Bill, Chris
   Chris, Dan                  Chris, Dan
   Chris, David                Chris, David
   David, Emily                David, Emily
                               Ann, Chris
                               Bill, Dan
                               Bill, David
                               Chris, Emily
                               Ann, Dan
                               Ann, David
                               Bill, Emily
```

In the fourth iteration, applying the recursive clause to this database results in one new ancestry fact being derived by matching a child in a parenthood fact with an ancestor in one of the new ancestry facts, as pictured below by the matching number:

```
isParentOf:                 isAncestorOf:
   Ann, Bill (1)               Ann, Bill
   Ann, Bob                    Ann, Bob
   Arnold, Bob                 Arnold, Bob
   Bill, Chris                 Bill, Chris
   Chris, Dan                  Chris, Dan
   Chris, David                Chris, David
   David, Emily                David, Emily
                               Ann, Chris
                               Bill, Dan
                               Bill, David
                               Chris, Emily
                               Ann, Dan
                               Ann, David
                            (1) Bill, Emily
```

The state of the database immediately after this fourth iteration is as shown below:

```
isParentOf:                        isAncestorOf:
   Ann, Bill                          Ann, Bill
   Ann, Bob                           Ann, Bob
   Arnold, Bob                        Arnold, Bob
   Bill, Chris                        Bill, Chris
   Chris, Dan                         Chris, Dan
   Chris, David                       Chris, David
   David, Emily                       David, Emily
                                      Ann, Chris
                                      Bill, Dan
                                      Bill, David
                                      Chris, Emily
                                      Ann, Dan
                                      Ann, David
                                      Bill, Emily
                                      Ann, Emily
```

Looking at these parenthood and ancestry facts, we see that the new fact added will not lead to any more ancestry facts because Ann does not appear as a child in a parenthood fact. Hence, if we applied a fifth iteration of the derivation rule, the database state would remain the same as it was in the previous state. This means that we have reached a fixedpoint of the computation, so the derivation rule has now been fully evaluated. If you look at the final population above, you will see that the 15 ancestry facts are precisely those that were obtained earlier by running the program. (The order in which the facts are displayed is irrelevant.)

Note that in the above description, the numbers in the second column were always adjacent to facts that had been added on the previous iteration. Hence, there was no need to recompute facts that had been already derived. By default, derivation rules without delta modifiers are evaluated incrementally by the LogiQL execution engine. This means that each subsequent execution, rather than recomputing from scratch, is driven by the changes made in the previous execution. This approach can significantly reduce the amount of effort required to complete the execution.

Exercise 3: In the map shown in Figure 3.2, nodes denote airports (BNE = Brisbane, KUL = Kuala Lumpur, BKK = Bangkok, LHR = London Heathrow, HEL = Helsinki), and the arrows denote direct (nonstop) flights.

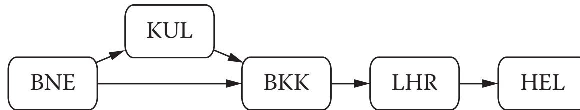

FIGURE 3.2 Direct flights between airports.

The following code is used to store direct flights and derive flight connections (direct or indirect):

```
Airport(a), hasAirportCode(a:c) -> string(c).
hasDirectFlightTo(a1, a2) -> Airport(a1), Airport(a2).
hasFlightTo(a1, a2) <- Airport(a1), Airport(a2).
hasFlightTo(a1, a2) <-
    hasDirectFlightTo(a1, a2) ;
    hasDirectFlightTo(a1, a3), hasFlightTo(a3, a2).
```

Using the iterative algorithm discussed above for evaluating recursive rules, specify the population of the predicate `hasFlightTo` at the end of each iteration.

UNIT 3.4: DELTA RULES AND PULSE PREDICATES

Previously, we have used delta modifiers to indicate that facts should be added to a database. In this unit we discuss other uses of delta modifiers in derivation rules to express changes to the database. Such uses are called **delta rules** to distinguish them from the **IDB rules** we have seen previously. We also discuss how delta rules can be used to record transitory changes or events using a particular kind of predicate called a **pulse predicate**.

For the examples in this unit, let us assume that persons may be identified simply by their names. Countries may be identified by their two-letter ISO country codes (e.g., “IE” for Ireland, “DE” for Germany, and “US” for the United States). The following program may now be used to record facts about persons and countries, as well as who is president of what country. The final constraint ensures that each person is president of at most one country at any given time:

```
Person(p), hasPersonName(p:pn) -> string(pn).
Country(c), hasCountryCode(c:cc) -> string(cc).
presidentOf[c] = p -> Country(c), Person(p).
presidentOf[c1] = p, presidentOf[c2] = p -> c1 = c2.
```

Delta Modifiers

Suppose the following assertions were made in 2008 to declare who was then president of Germany, Ireland, and the United States:

```
+presidentOf["IE"] = "Mary McAleese".
+presidentOf["DE"] = "Angela Merkel".
+presidentOf["US"] = "George W. Bush".
```

Recall that the delta modifier "+" indicates that the fact is to be added to the database. After adding the facts, the result of querying the `presidentOf` predicate is as follows:

```
US, George W. Bush
DE, Angela Merkel
IE, Mary McAleese
```

The compiler treats the above fact assertions as shorthand for longer assertions that populate not just the `presidentOf` predicate but also `Person`, `Country`, and their refmode predicates. For example, here is the result of querying `Person`:

```
George W. Bush
Angela Merkel
Mary McAleese
```

In January 2009, Barack Obama replaced George W. Bush to become the president of the United States, and in June 2010, Christian Wulff became the new president of Germany. As of May 2011, Mary McAleese was still president of Ireland. Suppose we now want to update the database accordingly. One way to do this would be to delete the presidency facts that no longer hold and then add the new presidency facts. To delete a fact, the delete modifier, denoted with a hyphen (`'-'`), is used. For example, the following code is used to retract two of the previous presidency facts:

```
-presidentOf["DE"] = "Angela Merkel".
-presidentOf["US"] = "George W. Bush".
```

Just as you may think of the "+" modifier as a plus sign for adding a fact, you may think of the "-" modifier as a minus sign for subtracting a fact. Querying `presidentOf` now displays just one fact:

```
IE, Mary McAleese
```

Note that deleting the presidency facts did not delete the `Person` or `Country` facts. For example, querying `Person` still returns the following:

```
George W. Bush
Angela Merkel
Mary McAleese
```

If you want to delete George W. Bush and Angela Merkel as instances of `Person` from the database, you will need to do that explicitly. For example, you may delete the two former presidents of Germany and the United States using the following fact retractions:

```
-Person("Angela Merkel").
-Person("George W. Bush").
```

A similar comment applies for countries. However, as we intend to add the new presidents for these countries, there is no point in deleting them. The two new president facts may be added as follows:

```
+presidentOf["US"] = "Barack Obama".
+presidentOf["DE"] = "Christian Wulff".
```

Querying `presidentOf` after these upserts displays the following:

```
US, Barack Obama
DE, Christian Wulff
IE, Mary McAleese
```

Note that combining the two deletions and two insertions just discussed in the same workspace update will not work. This is because LogiQL checks the validity of each delta rule individually with respect to the state of the workspace before any changes are made. That is, you cannot count on the order in which your assertions and retractions occur in your update. For example, placing the fact retractions before the assertions does not ensure that the deletions are done first. In the next unit, we will see how you can use LogiQL transactions to control the order in which workspace updates occur.

As a single-step way to replace a current value in a functional predicate with a new value, you may use the **upsert modifier**, depicted as a circumflex (`'^'`):

```
^presidentOf["US"] = "Barack Obama".
^presidentOf["DE"] = "Christian Wulff".
```

As the name *upsert* suggests, it may be used to either update or insert. If the key of the functional predicate already exists (e.g., "US" or "DE"), an update is performed, as in this example. Otherwise an insertion is made. Executing the above two upserts causes the `presidentOf` predicate to hold the following values:

```
US, Barack Obama
DE, Christian Wulff
IE, Mary McAleese
```

If you had not explicitly deleted George W. Bush and Angela Merkel as instances of `Person` from the database, then querying `Person` would now display:

```
George W. Bush
Angela Merkel
Mary McAleese
Christian Wulff
Barack Obama
```

Delta Logic

In the above, delta modifiers were used to unconditionally assert changes to a database. However, we might wish to make such changes conditionally; that is, we might want to use derivation rules for making such changes. In fact, delta modifiers may be used to qualify atoms in derivation rules, yielding what are called *delta rules*. In a delta rule, all head atoms must have delta modifiers. A delta rule such as `+p(x)<-+q(x)` can be interpreted as follows: If the database experiences an addition to the `q` predicate with argument `x`, then add a `p` fact with the same argument. Delta rules along with the assertions, retractions, and upserts described above together comprise a program's **delta logic**.

Pulse Predicates

One specialized form of delta rule makes use of pulse predicates. A pulse predicate is useful for asserting short-lived facts. Examples of such facts are those computed for the purpose of producing query results. Another use is for expressing one-time events, such as user interactions with the database through a graphical user interface (GUI).

A pulse predicate behaves as follows: For any given execution of the LogiQL engine, the predicate starts empty. A program may assert facts to

the predicate, but retractions are not allowed. Moreover, any assertions made during the execution are discarded at the end of the execution.

You have already seen one use of (unnamed) pulse predicates in the heads of queries. For example, consider the following query to list the presidents of countries other than the United States:

```
_(p) <- presidentOf[c] = p, !hasCountryCode(c:"US").
```

For the population given above, the query result is

```
Mary McAleese
Christian Wulff
```

Once this result is computed and displayed to the user, it is thrown away and is unavailable for later queries. That is, this **anonymous predicate**, denoted by the underscore, is a pulse predicate that temporarily holds a set of derived facts. If you want to persist these facts, you should give the predicate a name.

The other common use of pulse predicates is for capturing one-time events such as the pressing of a button on a screen form. As a simplified example, suppose that employee details are entered via instances of a Web form like the one shown in Figure 3.3. Forms like this may be created using GUI frameworks that support the creation and management of user interfaces. In this example, the framework includes a predefined entity type called *form* whose instances in a workspace are in one-to-one correspondence with instances of the form in the Web browser.

Components in the form have associated predicates in the workspace that hold their values. The above sample form includes three textfield components to capture the employee number, family name, and a given name of an employee. The form also includes a button (labeled “OK”) that the user presses to submit the details on the form to the LogiQL execution engine. The schema for the form could be declared as follows. (Note the

FIGURE 3.3 Screen form with submit button.

use of the colon [':'] after textField in the following code is similar to its earlier use with datetime and string. That is, it provides a name for a group of related predicates.)

```
textField:empNrOn[f] = n -> Form(f), int(n).
textField:familyNameOn[f] = fn -> Form(f), string(fn).
textField:givenNameOn[f] = gn -> Form(f), string(gn).
okButtonIsPressedOn(f) -> Form(f).
lang:isPulse[`okButtonIsPressedOn] = true.
```

Notice the use of the metapredicate lang:isPulse for declaring that okButtonIsPressedOn is a pulse predicate. The name of the pulse predicate is enclosed in brackets and preceded by the grave accent character ('`').

When the user enters the details and submits the form instance by pressing the OK button, we want to store the relevant data in the database. The part of the database schema used to capture the data could be declared as follows:

```
Employee(e), hasEmpNr(e:n) -> int(n).
familyNameOf[e] = fn -> Employee(e), string(fn).
givenNameOf[e] = gn -> Employee(e), string(gn).
```

The following code may now be used to copy the data entered on the form instance to the database in response to the pressing of the OK button:

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

The third line above references okButtonIsPressedOn. When the user enters the data and presses the OK button, the GUI framework asserts this fact, thereby invoking the execution of the rule to insert the relevant facts into the predicates appearing in the head of the rule. Once this is done, the pulse predicate okButtonIsPressedOn is effectively reset to be empty again.

The above example illustrates a common use of pulse predicates to implement event–action rules. In this case, the event is the pressing of a button, and the action is the updating of the database.

Tip: While insert (+) and delete (–) may be performed on any predicate (functional or non-functional), upsert (^) is allowed only on functional predicates.

Tip: Avoid applying retractions and insertions to the same functional predicate during the same update of a workspace. You cannot count on the order in which these actions are applied.

Tip: Indicate your intent to treat a predicate as a pulse predicate with the `lang:isPulse` metapredicate.

Tip: Pulse predicates may only be asserted and not retracted.

Exercise 4A: The following program (accessible as `Ex4a.logic`) is used to record the name and gender of various people, as well as what languages they speak and what languages they are fluent in. The last constraint is a subset constraint to ensure that people are fluent only in languages that they speak:

```
// Declarations
Person(p), hasPersonName(p:pn) -> string(pn).
Gender(g), hasGenderCode(g:gc) -> string(gc).
Language(lang), hasLanguageName(lang:ln) ->
    string(ln).
genderOf[p] = g -> Person(p), Gender(g).
speaks(p, lang) -> Person(p), Language(lang).
isFluentIn(p, lang) -> Person(p), Language(lang).
// Constraints
Person(p) -> genderOf[p] = _.
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
isFluentIn(p, lang) -> speaks(p, lang).
```

Compile this program and then populate it with the following data (accessible as `Ex4aData.logic`):

```
+genderOf["Norma"] = "F", +genderOf["Terry"] = "F".
+speaks("Norma", "English"), +speaks("Norma",
    "French").
+isFluentIn("Norma", "English").
+speaks("Terry", "English"), +speaks("Terry",
    "Japanese"),
```

```
+speaks("Terry", "Latin").
+isFluentIn("Terry", "English"), +isFluentIn("Terry",
     "Latin").
```

Query the predicate `genderOf`. It is now discovered that Terry is actually male, not female. Write delta logic to update Terry's gender accordingly, and again query the predicate `genderOf` to check that your update worked. The answer is in `Ex4aAnswer.logic`.

Exercise 4B: Now add a delta rule to the above program to ensure that if a fact is added that a person is fluent in a language, another fact will automatically be added indicating that person speaks that language. Test your code by then executing the following update (accessible as `Ex4bData.logic`), and querying the `speaks` predicate. *Hint*: Remember, you will need to have delta-modified atoms in both the head and body of the delta rule. The answer is in `Ex4bAnswer.logic`.

```
+isFluentIn("Norma", "Spanish").
```

Exercise 4C: Would it be possible to use the following IDB rule instead of the delta rule in the answer to the previous question? That is, what happens if a predicate contains both asserted and derived facts? Alter the above declarations and constraints to implement this approach. Edit the fact assertions to remove references to the `speaks` predicate. Print the `speaks` predicate and compare it to the previous results. Explain what has happened.

```
speaks(p, lang) <- isFluentIn(p, lang).
```

UNIT 3.5: TRANSACTION PROCESSING

We saw in the previous unit that during the execution of a rule, you could not count on the order in which changes are made to a workspace. Sometimes, of course, you want to control this order, and database designers have invented the idea of a **transaction** to organize such changes. In this unit we introduce transactions and describe at a high level their effect on a workspace. We then briefly look under the covers of transactions, to better understand how they are processed. Finally, we present a syntactic device, called a **stage suffix**, that gives you fine-grained control over how your data are handled during transaction processing.

Transactions

LogiQL workspaces contain both data (the facts we have asserted or derived) and program code (both what we have written and system code). Figure 3.4 provides a simplified, high-level picture of the activities that take place during the lifetime of a given workspace.

First we create the workspace. The newly created workspace contains a standard set of system predicates, but as yet has none of our program code. Once created, we can access and update the workspace through a series of transactions that either succeed or fail. Successful transactions update the workspace, whereas failing transactions leave the workspace in the state it was in before the transaction commenced.

In order to understand more thoroughly how transactions are processed, you need to be aware of two concepts: blocks and stages. A *block* is a unit of LogiQL code, typically originating in a file with a name ending in `.logic`. We have seen how, with the `lb` command, we can **install** code blocks into workspaces. In particular, we used either the `addblock` or the `exec` option to enter code. When the `addblock` option is used, the code becomes an active block, which means that it is automatically evaluated during every transaction. Conversely, the `exec` option can be used to request that a block should be evaluated exactly once. Such a block is a called an *inactive block*. More generally, your program has the ability to control the activated/deactivated status of installed blocks.

The LogiQL execution engine divides evaluation into two stages as illustrated in Figure 3.5.

The **initial stage** is used to process queries and provide on-demand evaluation of inactive blocks, such as you request with the `exec` option to `lb`. The updates requested by these blocks are tentatively accepted into the EDB, ignoring constraints.

In the **final stage**, first installed delta rules residing in active blocks are evaluated and checked against constraints. If no constraints are violated, the IDB rules in active blocks (previously installed or included in the transaction) are evaluated and checked against the constraints. If no constraints are violated during the execution of a transaction, the transaction is **committed**, and the tentative data updates are applied to the workspace. Otherwise, the transaction is **aborted** and the workspace is *rolled back* to its previous state (the state it was in just prior to the beginning of the transaction). In this sense, transactions are *atomic*, since either the whole of a transaction is accepted or none of it is.

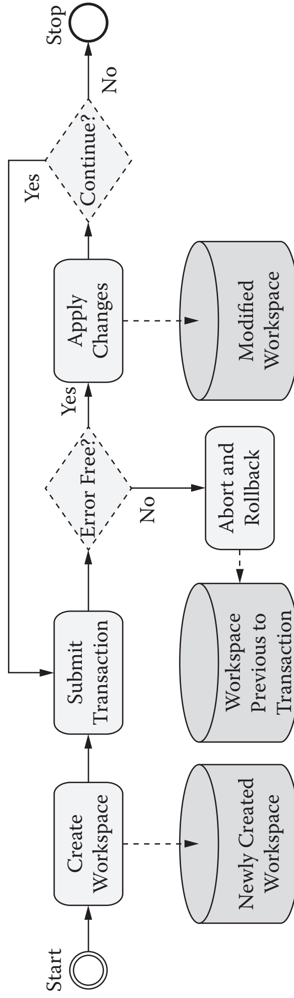

FIGURE 3.4 Workspace activity flow.

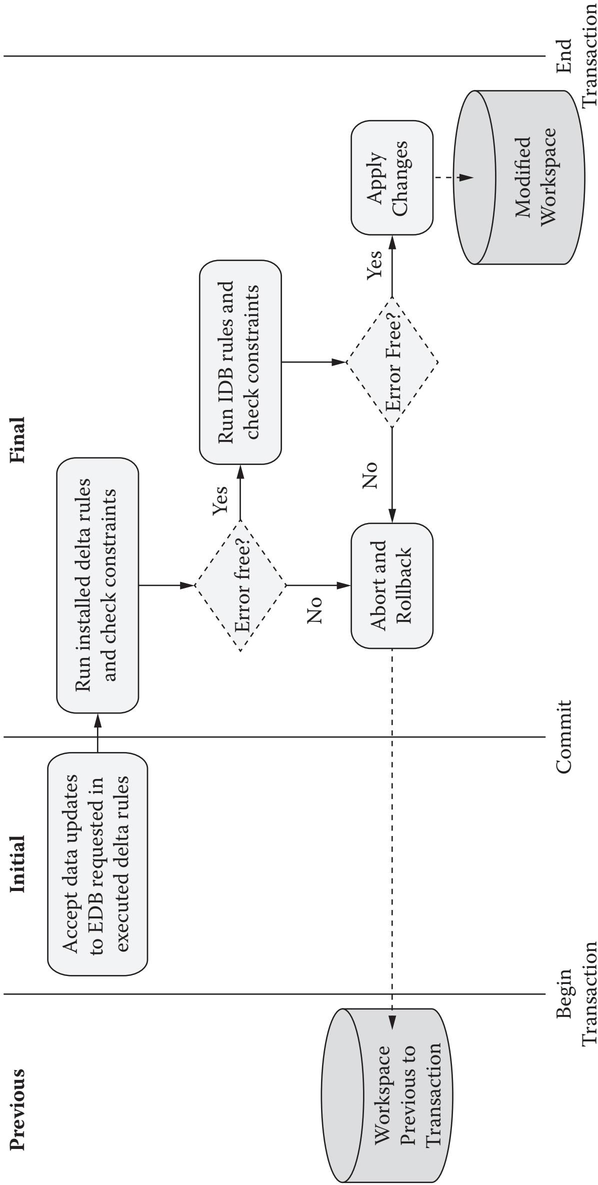

FIGURE 3.5 Transaction evaluation process.

We can continue this process with further transactions that install blocks of code or execute data updates until we are finished. At any moment of time between transactions, we are guaranteed that the state of the workspace represents the cumulative effect of applying all previous transactions, in the order in which they were committed.

Transaction Processing Example

To illustrate how transactions are processed, we will examine an example from the monarchy domain. To understand how the example is handled, an additional concept, **delta predicates**,* needs to be introduced. When you declare a new predicate, the LogiQL compiler automatically provides for you two additional predicates, called delta predicates, one each for recording the *requests* for adding and deleting data to the declared predicate. At the start of the initial stage of each transaction, these delta predicates are set to be empty. If the transaction includes a request to execute a data update against the declared predicate, the update is tentatively recorded into the associated delta predicates during the transaction's initial stage. During the final stage, the delta predicates are used to effect the actual changes made to the declared predicate.

For example, consider the code found in the file `transactions.lb`:

```
create --unique
addblock '// First transaction: install schema;
    see below'
exec '// Second transaction: assert data; see below'
echo "Printing Monarch"
print Monarch
echo "Querying isFemale:"
print isFemale
close --destroy
```

The file's first transaction installs the following code to declare the predicates and constraints:

```
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".
```

* Delta predicates should not be confused with delta rules, delta modifiers, or delta logic discussed earlier.

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

At the end of the first transaction, these predicates and their associated delta predicates are not populated, as shown in Table 3.6 for the `genderOf` and `isFemale` predicates. The same is true for the other predicates, but for simplicity let's follow the state of just the predicates shown here, focusing on the second transaction. Note that in this and subsequent tables in this section, the two delta predicates for these predicates are given names with prepended plus and minus signs. For example, the delta predicates for `genderOf` are labeled `+genderOf` and `−genderOf`.

The second transaction in the file includes the following 12 requests to insert data into the `genderOf` predicate:

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

At the beginning of the initial stage of the second transaction, the workspace is the same as at the end of the first transaction, with these predicates all empty. By the end of the initial stage of the second transaction, the data updates for `genderOf` are tentatively accepted, which we may picture as shown in Table 3.7. Though not shown here, the system also manages the implied updates for `Monarch`, and so on.

TABLE 3.6 Initial Predicates and Delta Predicates

genderOf	**+genderOf**	**-genderOf**	**isFemale**	**+isFemale**	**-isFemale**

TABLE 3.7 Predicates during the Initial Stage of the Second Transaction

genderOf	+genderOf	-genderOf
("Anne", "F")	("Anne", "F")	
("George I", "M")	("George I", "M")	
("George II", "M")	("George II", "M")	
("George III", "M")	("George III", "M")	
("George IV", "M")	("George IV", "M")	
("William IV", "M")	("William IV", "M")	
("Victoria", "F")	("Victoria", "F")	
("Edward VII", "M")	("Edward VII", "M")	
("George V", "M")	("George V", "M")	
("Edward VIII", "M")	("Edward VIII", "M")	
("George VI", "M")	("George VI", "M")	
("Elizabeth II", "F")	("Elizabeth II", "F")	

TABLE 3.8 isFemale Predicates during the Final Stage of the Second Transaction

isFemale	+isFemale	-isFemale
"Anne"	"Anne"	
"Victoria"	"Victoria"	
"Elizabeth II"	"Elizabeth II"	

TABLE 3.9 isMale Predicates during the Final Stage of the Second Transaction

isMale	+isMale	-isMale
"George I"	"George I"	
"George II"	"George II"	
"George III"	"George III"	
"George IV"	"George IV"	
"William IV"	"William IV"	
"Edward VII"	"Edward VII"	
"George V"	"George V"	
"Edward VIII"	"Edward VIII"	
"George VI"	"George VI"	

In the final stage of the second transaction the system evaluates the derivation rules, which in this case are isMale and isFemale. These predicates, which were empty at the start of the final stage, are now populated as in Tables 3.8 and 3.9.

At this point, the constraints are checked, and none are violated, so the transaction succeeds and changes to the program's predicates (not their associated delta predicates) are applied to the workspace.

The third transaction in the file interrogates the workspace about the `Monarch` and `isFemale` predicates:

```
echo "Printing Monarch"
print Monarch
echo "Querying isFemale:"
print isFemale
```

At the start of the third transaction, the `genderOf`, `isFemale`, and `isMale` predicates are populated as shown above, but their associated delta predicates are reset to be empty. The other predicates (`Monarch`, `hasMonarchName`, `Gender`, `hasGenderCode`) are also populated accordingly. The output produced by querying `isFemale` comprises the three `isFemale` facts in the above table.

Suppose that we now try to submit the following, fourth transaction. This would be a mistake, because Anne is already recorded to be a female, but let's see how the system would process this transaction:

```
echo "Adding a contradictory fact"
lb exec '+genderOf["Anne"] = "M".'
```

At the start of the initial phase, `genderOf` is populated as previously and its delta predicates are empty. At the end of the initial stage, the `genderOf` predicate has one more entry, and its delta predicates are populated as shown in Table 3.10.

TABLE 3.10 genderOf Predicates in the Initial Phase of Transaction Four

genderOf	**+genderOf**	**-genderOf**
("Anne", "F")	("Anne", "M")	
("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")		
("Anne", "M")		

When execution moves into the final stage and constraints are checked, it is found that the functional nature of the genderOf predicate is violated because Anne is now assigned two genders. Therefore, the fourth transaction is aborted. The information recorded in the program predicates and their delta predicates is sufficient for the system to determine the state of the workspace before the transaction began, simply by reversing the proposed changes. The workspace is now rolled back to that previous state.

Stage Suffixes

LogiQL provides some syntax that enables you to explicitly denote the various transaction stages of a predicate by qualifying its name with a **stage suffix**. For a specific predicate p, p@previous (or p@prev) references the population of the predicate as it was immediately before the start of the transaction, p@initial (or p@init) references the population of the predicate at the end of the initial stage, and p@final references the predicate in the final stage. In the absence of a suffix, the @final suffix is assumed.

A stage suffix can be valuable when you are trying to assess the impact of a new fact. Say, for example, that you wish to detect when someone's net worth has gone down. Here is a simple illustration:

```
Person(p), hasPersonName(p:n) -> string(n).
netWorth[p] = n -> Person(p), int(n).
status[] = s -> string(s).
^status[] = "Trouble City!!" <-
    ^netWorth[p] = _, netWorth[p] < netWorth@prev[p].
```

Each person has a name and a net worth. We also keep track of status with a unary predictate named status. If there is a change to the netWorth predicate such that the new net worth is less than the old, then we want the status to change. Let's assume that the workspace state is initialized as follows:

```
+status[] = "Okay".
+netWorth["Bob"] = 30.
```

Now, if we assert a new netWorth fact for Bob with a reduced value, we should expect the status predicate to change, which it does:

```
^netWorth["Bob"] = 10.
```

This approach for evaluating impact could be applied to audit changes coming into an application through a GUI. Alternatively, the above delta

rule could be converted into a constraint such that the negative change in net worth would cause the containing transaction to abort.

Summary

The processing of transactions is complex. Here is a summary of how the LogiQL execution engine handles them:

1. Computation is broken down into a sequence of atomic transactions, each of which is self-contained and leaves the workspace in a consistent state.
2. The processing of a transaction comprises two stages, initial and final. The initial stage begins with the same database state as the final state of the previous transaction.
3. The initial stage is responsible for recording fact assertion and retraction requests in delta predicates and handling queries. At the beginning of this stage, delta predicates are empty. If no delta rules exist, then this stage has no effect, other than producing the results of any queries.
4. The final stage is responsible for processing derivation rules of active blocks.
5. Derivation rule processing is made up of a series of steps, each of which may update the delta predicates. Steps continue until a fixed-point is reached.
6. At the end of the final stage, declared predicates are updated from delta predicates.

Tip: The `@previous` suffix may be used in the body of a rule but not in the rule head.

Exercise 5A: Alter the net worth example above to change the `status` message for any change (positive or negative) to an individual's net worth of more than 100%. Such a change might indicate that a data entry error had been made. *Hint:* You may wish to make use of the absolute value `abs` numeric function.

Exercise 5B: The following code (accessible as `grandparent.lb`) includes transactions to add program code for recording parenthood facts

and deriving grandparenthood facts, to execute some sample data, and to issue relevant queries:

```
create --unique
addblock '
  Person(p), hasPersonName(p:pn) -> string(pn).
  isParentOf(p1, p2) -> Person(p1), Person(p2).
  isGrandparentOf(p1, p2) -> Person(p1), Person(p2).
  isGrandparentOf(p1, p2) <- isParentOf(p1, p3),
    isParentOf(p3, p2).
'
exec '
  +isParentOf("Ivor", "Norma").
  +isParentOf("Norma", "Linda").
  +isParentOf("Norma", "David").
  +isParentOf("Terry", "Linda").
  +isParentOf("Terry", "David").
  +isParentOf("David", "Emily").
  +isParentOf("David", "Sam").
'
echo "Querying isParentOf:"
print isParentOf

echo "Querying isGrandparentOf:"
print isGrandparentOf

close --destroy
```

The program code is fine so long as our knowledge of relevant parenthood facts is complete. However, suppose instead that our knowledge is incomplete, and that we want to record a known grandparenthood fact even if we do not know the parenthood facts from which it could be derived. For example, we might know that Ivor is a grandparent of Graham without knowing which of Ivor's children is a parent of Graham. In this case, we need to assert the fact that Ivor is a grandparent of Graham directly, such as by executing the following fact assertion:

```
+isGrandparentOf("Ivor", "Graham").
```

Explain what happens if you make this assertion. How would you fix the problem?

Exercise 5C: One way we might try to deal with this situation is to use delta rules instead of IDB rules to automatically add grandparent facts when relevant parenthood facts are added. Copy the file `grandparent2.lb` and modify its program code to test this new approach. For this question, you may ignore any need to provide delta rules to manage fact retractions.

Exercise 5D: An alternative approach to that of Exercise 5C for dealing with incomplete knowledge of parenthood is to use an EDB predicate `asserted_isGrandparentOf` for asserting grandparenthood facts, an IDB rule `derived_isGrandparentOf` to derive grandparenthood from parenthood, and then use the disjunction of these two rules to derive `isGrandparentOf`, whose population is the union of the facts in the other two predicates. Copy the file `grandparent4.lb` and modify its program code to test this new approach.

UNIT 3.6: ADDITIONAL BUILT-IN OPERATORS AND FUNCTIONS

Previous units have discussed various arithmetic operators (e.g., +, –, *), comparison operators (e.g., <, >, =, < =, > =, ! =), logical operators (e.g., !, comma, ;), scalar functions (e.g., `divide`), and aggregation functions (`count`, `total`, `min`, and `max`). This unit discusses some further built-in operators and functions that you may find useful.

Arithmetic

LogiQL includes a variety of mathematical functions for manipulating numeric data. As a simple example, the following program (`maths.lb`) uses `abs` to compute the absolute value of a number and `sqrt` to compute the square root. Note that while the `absoluteValueOfTemp` predicate contains two facts, the `sqrtOfTemp` has only one. Can you think why this might be?

One of the two temperatures is negative; its square root is undefined:

```
create --unique
addblock ws '
  Temperature(t), hasCelsiusValue(t:cv) -> float(cv).
  absoluteValueOfTemp[t] = at -> Temperature(t),
    float(at).
  absoluteValueOfTemp[t] = at <-
    hasCelsiusValue(t:cv), abs[cv] = at.
```

```
  sqrtOfTemp[t] = st -> Temperature(t), float(st).
  sqrtOfTemp[t] = st <- hasCelsiusValue(t:cv),
    sqrt[cv] = st.
'
exec '
  +Temperature(25f), +Temperature(-20f).
'
echo "Querying absoluteValueOfTemp:"
print absoluteValueOfTemp
echo "Querying sqrtOfTemp:"
print sqrtOfTemp
close --destroy
```

The query output is as follows

```
Querying absoluteValueOfTemp:
25, 25
-20, 20
Querying sqrtOfTemp:
25, 5
```

String Manipulation

LogiQL also includes operators and functions that are handy for `string` manipulation. For example, you can use the *string concatenation* operator (`'+'`) to paste together two `string` arguments. The following program (`concatNames.lb`) uses + to derive the full name of employees by concatenating their given name to a space character and then to their family name:

```
create --unique
addblock '
  Employee(e), hasEmpNr(e:n) -> int(n).
  givenNameOf[e] = gn -> Employee(e), string(gn).
  familyNameOf[e] = fn -> Employee(e), string(fn).
  Employee(e) -> givenNameOf[e] = _,
    familyNameOf[e] = _.
  fullNameOf[e] = pn -> Employee(e), string(pn).
  fullNameOf[e] = pn <- pn = givenNameOf[e] + " " +
    familyNameOf[e].
'
exec '
  +givenNameOf[1] = "John", +familyNameOf[1] =
    "Smith".
```

```
  +givenNameOf[2] = "Eve", +familyNameOf[2] = "Jones".
'
echo "Querying fullNameOf:"
print fullNameOf
close --destroy
```

When run, the query output is as follows:

```
Querying fullNameOf:
  2, Eve Jones
  1, John Smith
```

The `string:like` function is handy for string pattern matching. It has the form `string:like(str,pattern)` where `str` is a string expression and `pattern` is a quoted string that may include the wildcard characters (' _ '), meaning any single character, and ('%'), meaning any sequence of zero or more characters. The following program (`courses.lb`) demonstrates its use for finding the computer science courses (those courses whose course codes begin with "CS") and the entry-level courses (those courses whose course codes have "1" as their third character):

```
create --unique
addblock '
  Course(c), hasCourseCode(c:cc) -> string(cc).
  ComputerScienceCourse(c) -> Course(c).
  ComputerScienceCourse(c) <- hasCourseCode(c:cc),
    string:like(cc, "CS%").
  EntryLevelCourse(c) -> Course(c).
  EntryLevelCourse(c) <- hasCourseCode(c:cc),
    string:like(cc, "__1%").
'
exec '
  +Course("CS100"), +Course("CS200"),
    +Course("MA100").
'
echo "Querying ComputerScienceCourse:"
print ComputerScienceCourse
echo "Querying EntryLevelCourse:"
print EntryLevelCourse
close --destroy
```

The query results are as shown below:

```
Querying ComputerScienceCourse:
  CS100
  CS200
Querying EntryLevelCourse:
  CS100
  MA100
```

To illustrate some other string manipulation functions, consider the problem of extracting a person's last name when given the person's full name. To make things interesting, the last name may contain several parts, such as "Marilyn vos Savant." Here is how you might accomplish this task, assuming the full names are given to you in the predicate `hasPersonName`:

```
lastName(s) -> string(s).
// Extract the last name from a Person's full name
lastName(s) <-
    hasPersonName(_:n),
    f = string:length[string:split[n, " ", 0]],
    // Length of first name
    t = string:length[n], // Length of full name
    string:substring[n, f + 1, t - f - 1] = s.
```

Several other `string` functions are used to perform the extraction. The `string:length` function is used twice, once to compute the length of the first name and once to compute the total name length. Its argument is the `string` whose length is desired.

The person's first name is extracted using `string:split` to break out that part of the full name from its start up until the first space character. Its arguments are the string to be split, the character upon which the split is based, and an index into the resulting segment. The result of `string:split` is the specified segment of the argument `string`.

Finally, the last name is extracted using the `string:substring` function, beginning with the character after the first space and continuing for a number of characters computed by subtracting off the length of the first name.

We will see this code segment again in the next section when used to demonstrate some of the LogiQL functions for ordering data.

TABLE 3.11 IQs of Famous People

Person	IQ
Albert Einstein	160
Bill Gates	160
Dwight Eisenhower	122
Marilyn vos Savant	228
Hillary Clinton	140
Nicole Kidman	132

Aggregation Functions for Ordering

We now discuss two more aggregate functions useful for ordering the facts in a predicate. Recall Table 3.11 of IQs considered in Unit 2.6.

Suppose we want to see these values in ascending order. Such *ranking* queries assign numeric ranks to the objects in an ordered list. For this to be possible, the values must be based on one of LogiQL's primitive types, for which an ordering relation (e.g., less than, <) is ensured.

LogiQL provides two aggregation functions for ranking the facts in a predicate based on the values of one of the predicate's roles: `seq` generates ranks suitable for indexing into the facts according to the order of the role's values; `list` enables navigation through the predicate's facts from least to greatest in single steps. As with other aggregation functions, the data that is being ranked can be computed using a condition.

Aggregation rules using the `seq` function take the following form, where `xValueOfRank` is replaced by the name of a ranking predicate that associates each rank `r` with the `x` value that has that rank. `Cx` is the condition that provides the data being ranked.

```
xValueOfRank[r] = x <- seq<<r = x>> Cx.
// For each rank r, return the x value of that rank ,
// where Cx is true.
```

Note that since the ranking predicate is populated with a set of facts as a result of this derivation, the `seq` "function" is not a function in the usual sense (where a single value is returned), unlike the aggregation functions considered previously (`count`, `total`, `min`, and `max`).

As an example of the use of `seq`, the following program (taken from `IQtop.lb`) returns the IQ values from the data in the table in

ascending order. First, the `Person` entity is declared along with a property predicate providing IQs for `Persons` and corresponding constraints:

```
Person(p), hasPersonName(p:pn) -> string(pn).
iqOf[p] = iq -> Person(p), int(iq).
iqOf[_] = iq -> iq < = 250. // Each IQ is at most 250.
Person(p) -> iqOf[p] = _.   // Each person has an IQ.
```

We can populate these predicates with data as follows:

```
+iqOf["Albert Einstein"] = 160.
+iqOf["Bill Gates"] = 160.
+iqOf["Dwight Eisenhower"] = 122.
+iqOf["Marilyn vos Savant"] = 228.
+iqOf["Hillary Clinton"] = 140.
+iqOf["Nicole Kidman"] = 132.
```

If we query `iqOf`, the following results are as shown here:

```
Nicole Kidman, 132
Hillary Clinton, 140
Marilyn vos Savant, 228
Dwight Eisenhower, 122
Bill Gates, 160
Albert Einstein, 160
```

Now, we would like to see the IQ scores in ascending order. To do this, we first separate out the IQ scores into a separate predicate, `iqScores`.:

```
iqScores(iq) -> int(iq).
iqScores(iq) <- iqOf[_] = iq.
```

We will now use `seq` to compute the rankings and store them into the `iqOfRank` functional predicate:

```
iqOfRank[i] = iq -> int(i), int(iq).
iqOfRank[i] = iq <- seq<<i = iq>> iqScores(iq).
```

A query of `iqOfRank` then produces the following results:

```
0, 122
1, 132
```

```
2, 140
3, 160
4, 228
```

There are several ways that we can extend this simple example to deal with more interesting situations. Say we wanted to know not only what the order of scores was, but which people had those scores. This is easily done by adding another predicate, `orderedIQof` that combines the ordering information with the original `iqOf` predicate:

```
orderedIQof[p, iq] = i -> Person(p), int(iq), int(i).
orderedIQof[p, iq] = i <- iqOfRank[i] = iq,
    iqOf[p] = iq.
```

The result of printing `orderedIQof` is the following. Note, in particular, that the actual rankings are specified in the third column and not in the order in which the facts are displayed:

```
"Nicole Kidman" 132 1
"Marilyn vos Savant" 228 4
"Bill Gates"      160 3
"Albert Einstein" 160 3
"Hillary Clinton" 140 2
"Dwight Eisenhower" 122 0
```

Other variants you might find useful are to sort the facts in descending order or to see just a subset of the values. Here are the addition rules you would use to compute just the largest three values of IQ data:

```
numberOfIQs[] = n -> int(n).
numberOfIQs[] = n <- agg<<n = count()>>
    iqOfRank[i] = _.
// The number of different IQ values.

topThreeIQvalues[i] = iq -> int(i), int(iq).
topThreeIQvalues[i] = iq <- numberOfIQs[] = n ,
    iqOfRank[j] = iq, j > = n - 3, i = n - j.
// The top three IQ values.
```

Another way of accessing ordered data is using a linked list. The `list` aggregation supports this approach by populating two predicates. The first, unary, predicate holds the head of the list, and the second, binary, predicate holds pairs containing an element of the list as well as the next element.

Here is an example of using `list` to compute a report of IQs in alphabetical order of the person's last name. The example also illustrates use of some of the `string` functions described the earlier in this unit.

The example reuses the same basic schema as the previous example:

```
Person(p), hasPersonName(p:s) -> string(s).
iqOf[p] = iq -> Person(p), int(iq).
iqOf[_] = iq -> iq < = 250. // Each IQ is at most 250.
Person(p) -> iqOf[p] = _.  // Each person has an IQ.
```

It also reuses the rule we developed in the last section for extracting last names:

```
lastName(s) -> string(s).
// Extract the last name from a Person's full name
lastName(s) <- hasPersonName(_:n),
    f = string:length[string:split[n, " ", 0]],
    // Length of first name
    t = string:length[n], // Length of full name
    string:substring[n, f + 1, t - f - 1] = s.
```

Once the last names have been extracted, they can be ordered using the `list` aggregation function:

```
head(n) -> string(n).
// The head node of a linked list.
next(n1, n2) -> string(n1), string(n2).
// The source-target links in a linked list.
head(n1), next(n1, n2) <- list<< >> lastName(n1).
// Store the sorted last names into a linked list.
```

In the above snippet, two predicates (`head` and `next`) are defined. `Head` holds the alphabetically least last name, and `next` holds pairs of last names such that the first element of the pair is immediately succeeded, in alphabetical order, but the second `list<< >>` is used to simultaneously populate these two predicates, taking its source data from the `lastName` predicate.

To produce the output report, the nodes of the linked list are visited in order, each providing the data to produce a segment of the report. The segments are pasted together to produce the final report:

```
visit[n] = s -> string(n), string(s).
// Traverse the linked list
```

```
visit[n1] = s <- next(n1, n2), s = format[n1] +
    visit[n2].
// Format head and recursively concatenate to the
// remainder
visit[n] = s <- next(_, n), !next(n, _), s = "".
// The last element must be in the list and not have a
// successor
```

Traversal is accomplished with the `visit` function, which, for each node visited, recursively computes the report for the contents of the linked list from that node on, concatenating the results using the `string` addition function (`'+'`).

`Visit` makes use of a utility function, `format`, for retrieving the IQ information associated with a given last name and formatting the results. Note that for this example, `format` assumes that last names are unique. Also, `format` makes use of the `string:like` and `string:convert` predicates described above:

```
format[n] = s -> string(n), string(s).
// Format one line of the output report
format[n] = s <- lastName(n), hasPersonName(p:fn),
    string:like(fn, "% " + n),
    iqOf[p] = iq,
    int:string:convert[iq] = siq,
    s = fn + ": " + siq + "\n".
```

It remains only to start off the visiting process on the first element of the linked list:

```
report(s) -> string(s).
report(s) <- head(n), s = visit[n].
```

The code for this example can be found in the file `IQlist.lb`.

Tip: Become familiar with the full set of available built-in functions so you can make use of them when the occasion arises.

Exercise 6A: Table 3.12 (available in `USCitiesData.logic`) stores name and population data for various major cities in the United States. Population figures are for the year 2010. Although cities are primarily

TABLE 3.12 City Populations

City Number	City Name	State Name	Population
1	Cleveland	Ohio	396,815
2	Columbus	Georgia	189,885
3	Columbus	Ohio	787,033
4	New York	New York	8,175,133
5	Portland	Maine	66,194
6	Portland	Oregon	583,776
7	Los Angeles	California	3,792,621
8	San Diego	California	1,307,402
9	San Jose	California	945,942
...	...	...	...

identified here by a city number, the combination of their city name and state is also unique.

A basic schema for this example is shown below:

```
City(c), hasCityNr(c:n) -> int(n).
State(s), hasStateName(s:sn) -> string(sn).
cityNameOf[c] = cn -> City(c), string(cn).
stateOf[c] = s -> City(c), State(s).
populationOf[c] = n -> City(c), int(n).
City(c) -> cityNameOf[c] = _, stateOf[c] = _,
    populationOf[c] = _.
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.
```

The file `USCities.lb` includes this schema and extends it with the following declaration and rule stub, as well as the relevant data and query:

```
extendedCitynameOf[c] = ecn -> City(c), string(ecn).
extendedCitynameOf[c] = ecn <-
// *** supply the rule body here
```

The derived predicate `extendedCitynameOf` is intended to store the extended name of cities, where an extended name consists of the city name and its state name, separated by a comma. Complete the derivation rule

for this predicate and run the program. The expected output from querying this predicate is as follows:

```
Querying extendedCitynameOf:
  9, San Jose, California
  8, San Diego, California
  7, Los Angeles, California
  6, Portland, Oregon
  5, Portland, Maine
  4, New York, New York
  3, Columbus, Ohio
  2, Columbus, Georgia
  1, Cleveland, Ohio
```

Exercise 6B: Add a derivation rule to your solution for `endsInLand(cn)` to determine each city name `cn` that ends in the characters `"land"`.

Exercise 6C: Add code to your solution to derive the names of the cities having the three highest populations.

UNIT 3.7: CONSOLIDATION EXERCISE 3

This exercise gives you a chance to test how well you have mastered the new topics covered in this chapter. As much of the content of this chapter has been theoretical and wide ranging, the questions are less integrated than in previous chapters.

Q1: Explain what is wrong with the following code, and revise it to fix the problem. The answer can be found in the file `Q1Answer.logic`.

```
Customer(c), customerNR(c:n) -> int(n).
Vehicle(v), vehicleVIN(v:n) -> string(n).
Accessory(a), accessorySerialNr(a:n) -> string(n).
purchased(c, v) -> Customer(c), Vehicle(v).
purchased(c, a) -> Customer(c), Accessory(a).
```

Q2: Given the following declarations, correctly classify each of the rules below as safe or unsafe. The answers can be found in the file `Q2Answer.logic`:

```
Person(p), hasPersonName(p:pn) -> string(pn).
Book(b), hasISBN(b:isbn) -> string(isbn).
```

```
smokes(p) -> Person(p).
isPairedWith(p1, p2) -> Person(p1), Person(p2).
isOdd(n) -> int(n).
isLonely(p) -> Person(p).
```

(a) `isNonSmoker(p) <- !smokes(p).`

(b) `isOdd(n) <- n = 1 ; n = 3.`

(c) `isPairedWith(p1, p2) <- smokes(p1) ; smokes(p2).`

(d) `isPairedWith(p1, p2) <- smokes(p1), smokes(p2).`

(e) `isPairedWith(p1, p2) <- !smokes(p1), !smokes(p2).`

(f) `isLonely(p) <- Person(p), !isPairedWith(p, _).`

Q3: The file `ages.lb` includes transactions to add the following schema, execute the data shown, and query the `ageOf` and `isaSenior` predicates:

```
// Schema
Person(p), hasPersonName(p:pn) -> string(pn).
ageOf[p] = n -> Person(p), int(n).
isaSenior(p) -> Person(p).

// Data
+ageOf["Elvis"] = 42.
+ageOf["Terry"] = 64.
+ageOf["Walter"] = 90.
+ageOf["Xena"] = 90.
+isaSenior("Walter").
+isaSenior("Xena").
+isaSenior("Methuselah").
```

(a) Write a delta rule to insert the fact that a person is a senior if a fact is added or updated that the person has an age of at least 65. Test this by writing other delta rules to insert the fact that Bertie is aged 97 and to update Terry's age to 65.

(b) Starting again from `ages.lb`, write delta rules to retract the fact that a person is a senior if that person's age is changed to a value below 65 or that person's age is deleted. Test this by updating Xena's age to 30 and deleting Walter's age fact.

(c) Starting again from `ages.lb`, write a delta rule that will delete the fact that a person exists if the fact for that person's age is deleted. Test this by writing another delta rule to delete Xena's age fact.

(d) Suppose that instead of the delta rule approach used in (b), the following derivation rule is used: `isaSenior(p)<-ageOf(p)> = 65.` Discuss the advantages and disadvantages of this alternative approach.

The answers can be found in the file `Q3Answer.logic`.

Q4: Devise a rule for computing the number of descendants of a British monarch. You can make use of the schema provided in `foo.lb` and the parenthood information in `bar.lb`. *Hint:* You will need to combine aggregation and recursion. Normally LogiQL does not allow this because of the danger of infinite loops. However, you can instruct the compiler to allow you to do this by adding the following metarule to your code:

```
lang:compiler:disableWarning:AGGREGATE_RECURSION[]
    = true.
```

Of course, you should be careful when doing this that your code is, in fact, guaranteed to terminate. An answer can be found in `Q4Answer.logic`.

Q5: The table presented in Exercise 3.6.5 was used to illustrate aggregation functions for ordering data, stores name and population data for various major cities in the United States. The files `USCities5.logic` and `USCitiesData.logic` contain the schema and city population data describing it that are used in the next several exercises. We wish to produce a report describing the population of each city. Here is an appropriate predicate declaration:

```
populationStatementOf[c] = psc -> City(c), string(psc).
```

Provide an IDB rule to populate this predicate. *Hint:* The built-in type conversion predicate `int:string:convert[n]=s` converts the value of the integer expression `n` to a character string `s`. Your output should look like the following:

```
Querying populationStatementOf:
9, The population of San Jose, California = 945942
8, The population of San Diego, California = 1307402
7, The population of Los Angeles, California = 3792621
6, The population of Portland, Oregon = 583776
```

```
5, The population of Portland, Maine = 66194
4, The population of New York, New York = 8175133
3, The population of Columbus, Ohio = 787033
2, The population of Columbus, Georgia = 189885
1, The population of Cleveland, Ohio = 396815
```

The answer can be found in the file `Q5Answer.logic`.

Q6: Use the `max` and `min` aggregation functions to extend the program in `USCities5.logic` to derive the following: **(a)** the highest population; **(b)** the lowest population; **(c)** the extended name of the city/cities with the highest population; **(d)** for each state, the name of the city with the highest population for that state (based on the data supplied). The answers are provided in the file `Q6Answer.logic`.

Q7: Extend the file `USCities5.lb` with code to derive the extended name of the city with the seventh highest population (assuming no ties in the population figures). The answer is provided in the file `Q7Answer.logic`.

Q8: Consider the following three rules, in which a given baseball player has a batting average expressed in the property predicate `hasAverage`. Also, players are considered *heavy hitters* if their batting averages are greater than 0.300.

(a) Installed derivation rule:

```
heavyHitter(p) <- playerHasAverage[p] > .300.
```

(b) Installed delta rule:

```
+heavyHitter(b) <- +playerHasAverage[b] > .300.
```

(c) Executed delta rule:

```
+heavyHitter(b) <- playerHasAverage[b] > .300.
```

Provide a comment for each rule describing its behavior. *Hint:* Each of the three has a different behavior. The answer can be found in the file `Q8Answer.logic`.

ANSWERS TO EXERCISES

Answer to Exercise 1A:

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

maxExpense[] = n -> decimal(n).
maxExpense[] = n <- agg<<n = max(e)>>
    expenseOf[_] = e.
// The maximum expense is the largest expense.

isMostExpensive(i) -> Item(i).
isMostExpensive(i) <- expenseOf[i] = maxExpense[].
// Item is most expensive if it has the greatest
// expense.
```

Result:

```
/— - start of isMostExpensive facts— -\
 Accommodation
 Travel
\— — end of isMostExpensive facts— — /
```

Answer to Exercise 1B:

```
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.

maxExpenseOf[c] = n -> Claim(c), decimal(n).
maxExpenseOf[c] = n <- agg<<n = max(e)>>
  claimItemExpense[c, _] = e.
// The maximum expense of claim c is the
// the greatest single expense amount on claim c.

hasMostExpensiveItem(c, i) -> Claim(c), Item(i).
hasMostExpensiveItem(c, i) <- claimItemExpense[c, i] =
    maxExpenseOf[c].
// Claim c has most expensive item i if
// the expense of item i is the max expense on
    claim c.
```

Result:

```
/— - start of hasMostExpensiveItem facts— -\
   2, Travel
   1, Travel
   1, Accommodation
\— — end of hasMostExpensiveItem facts— — /
```

Answer to Exercise 2A:

The derivation rule for `isOff` is unsafe because its head variable does not occur in a positive context in the rule body. To fix the rule, modify it as shown:

```
isOff(l) <- Light(l), !isOn(l).
// A light is off it is a light that is not on.
```

Answer to Exercise 2B:

```
p(x, y, z) <- q(x), !r(y), z > 1.
```

The head variable `y` does not appear in a positive context in the body. The head variable `z` does not appear in a positive context of a domain predicate in the body:

```
p(x, y, z) <- q(x), r(y), s(z) ; q(y), r(z).
```

The second disjunct includes only two of the three head variables.

Note that the *and* operator has priority over the *or* operator, so the above disjunctive rule is equivalent to the following:

```
p(x, y, z) <- (q(x), r(y), s(z)) ; (q(y), r(z)).
```

Answer to Exercise 3:

```
hasFlightTo (Iteration 1):
BNE, KUL
BNE, BKK
KUL, BKK
BKK, LHR
LHR, HEL

hasFlightTo (Iteration 2):
BNE, KUL
BNE, BKK
```

```
KUL, BKK
BKK, LHR
LHR, HEL

hasFlightTo (Iteration 3):
BNE, KUL
BNE, BKK
KUL, BKK
BKK, LHR
LHR, HEL
BNE, LHR
KUL, LHR
BKK, HEL
```

Answer to Exercise 4A:

The update may be done in a single step: thus,

```
^genderOf["Terry"] = "M".
```

Consider the alternative of first executing this retraction:

```
-genderOf["Terry"] = "F".
```

and later executing this assertion:

```
+genderOf["Terry"] = "M".
```

What goes wrong if you try this approach? *Answer*: During the period between the retraction and the assertion, `"Terry"` will not have an associated `Gender`. This violates the first constraint, that all `Persons` have `Genders`.

Answer to Exercise 4B:

```
+speaks(p, lang) <- +isFluentIn(p, lang).
```

After asserting that Norma is fluent in Spanish, a query of `speaks` displays the following:

```
Terry, English
Norma, English
Terry, Latin
Terry, Japanese
```

```
Norma, French
Norma, Spanish
```

Answer to Exercise 4C:

Results of first query:

```
Terry, English
Norma, English
Terry, Latin
Terry, Japanese
Norma, French
```

Result of second query:

```
Terry, English
Norma, English
Terry, Latin
```

A predicate in LogiQL can either be an IDB predicate or an EDB predicate, but not both. Note that in the IDB version two facts have disappeared.

In the next unit, we discuss how to write a *cascading* delta rule for `speaks` to deal with retractions, while still preserving the subset constraint from fluency to speaking.

Answer to Exercise 5A:

The additional rule is shown below. A file to test the program with relevant updates is accessible as `netWorth.lb`:

```
^status[] = "Big Changes Happening!!" <-
^netWorth[p] = _,
 abs[netWorth[p] - netWorth@prev[p] /
     netWorth@prev[p]] >= 2.
```

Answer to Exercise 5B:

`isGrandparentOf` is an IDB predicate; trying to directly assert a fact into it, which is an EDB operation, raises an exception. That is, a predicate cannot be part of both the IDB and the EDB.

One way to get around this limitation is to break the grandparent relation into two parts, one for the IDB and one for the EDB:

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

```
isGrandparentOfEDB(p1, p2) -> Person(p1), Person(p2).
isGrandparentOfIDB(p1, p2) <-
    isGrandparentOfEDB(p1, p2).
```

You can then add the new fact into the EDB predicate as follows:

```
+isGrandparentOfEDB("Ivor", "Graham").
```

During the initial stage of the transaction, the `isGrandparentOfEDB` fact is added. Then, during the final stage, the second `isGrandparentOfIDB` rule fires, adding the new fact into the IDB predicate.

Answer to Exercise 5C:

The delta rules to deal with assertions are shown below. A file to test the program with relevant updates is accessible as `grandparent3.lb`:

```
+isGrandparentOf(p1, p2) <- +isParentOf(p1, p3),
     isParentOf(p3, p2).
+isGrandparentOf(p1, p2) <- isParentOf(p1, p3),
     +isParentOf(p3, p2).
```

Note: If you wish grandparenthood facts to be deleted when parenthood facts that imply them are deleted, delta rules to cater for such fact retractions are also needed. As the answer to the next question shows, use of IDB rules is typically preferable to writing sets of delta rules for such a case, since the maintenance of data updates is then managed for you automatically.

Answer to Exercise 5D:

The program including IDB rules is shown below. A file to test the program with relevant updates is accessible as `grandparent5.lb`:

```
Person(p), hasPersonName(p:pn) -> string(pn).
isParentOf(p1, p2) -> Person(p1), Person(p2).
asserted_isGrandparentOf(p1, p2) -> Person(p1),
    Person(p2).
derived_isGrandparentOf(p1, p2) -> Person(p1),
    Person(p2).
derived_isGrandparentOf(p1, p2) <-
isParentOf(p1, p3), isParentOf(p3, p2).
isGrandparentOf(p1, p2) -> Person(p1), Person(p2).
    isGrandparentOf(p1, p2) <-
```

```
  asserted_isGrandparentOf(p1, p2) ;
  derived_isGrandparentOf(p1, p2).
```

Answer to Exercise 6A:

```
extendedCitynameOf[c] = ecn -> City(c), string(ecn).
extendedCitynameOf[c] = ecn <- stateOf[c] = s,
    hasStateName(s:sn),
    ecn = cityNameOf[c] + ", " + sn.
```

The full program is accessible as `USCities2.lb`.

Answer to Exercise 6B:

```
endsInLand(cn) <- cityNameOf[_] = cn,
    string:like(cn, "%land").
```

The relevant output is shown below. The full program is accessible as `USCities3.lb`:

```
Querying citynames ending in 'land':
  Cleveland
  Portland
```

Answer to Exercise 6C:

```
population(i) -> int(i).
population(i) <- populationOf[_] = i.

populationOfRank[i] = pop -> int(i), int(pop).
populationOfRank[i] = pop <- seq<<i = pop>>
    population(pop).

numberOfCities[] = n -> int(n).
numberOfCities[] = n <- agg<<n = count()>> City(_).

threeMostPopulatedCities[i] = s -> int(i), string(s).
threeMostPopulatedCities[i] = s <-
    n = numberOfCities[],
    j > = n - 3,
    i = n - j,
    populationOfRank[j] = pop,
    populationOf[p] = pop,
    cityNameOf[p] = s.
```

The relevant output is shown below. The full program is accessible as `USCities4.lb`.

```
Querying threeMostPopulatedCities:
  1 "New York"
  2 "Los Angeles"
  3 "San Diego".
```