Appendix G: LogiQL and Predicate Logic

LogiQL is a programming language capable of computing correct answers for properly formulated problems. It has evolved from efforts to apply formal logical reasoning to databases, and this appendix describes the relationship between logic and LogiQL. In particular, the appendix describes how LogiQL is related to propositional logic and first-order predicate logic.

Propositions are asserted by declarative sentences and are always true or false but not both. In *propositional logic*, atomic propositions are denoted by propositional constants (e.g., R = "It is raining"; S = "It is snowing"). Compound propositions are formed by applying propositional operators to other propositions, using parentheses if needed. For example, using ~ for the negation operator and ∧ for the conjunction operator, we could use the formula ~(R∧S) to denote the proposition that "It is not both raining and snowing." Results about propositional patterns are indicated by using propositional variables (e.g., p, q) to stand for any propositions in general. For example, ~~p is equivalent to p regardless of which proposition is substituted for p.

First-order logic (*FOL*), also called *first-order predicate logic*, *predicate calculus, or quantification theory*, extends propositional logic with predicates, quantifiers, and individual constants. In logic, an *individual* is any individual object (entity or value). For a given universe of discourse, specific individuals are denoted by individual constants (e.g., terry), and specific predicates are denoted by predicate constants with their arguments in parentheses, for example, isTallerThan(terry,norma). General results may be stated using individual variables (to range over any individual) and quantifiers. The universal quantifier ∀ means "for each" or "for all." For example, the formula ∀x(~isTallerThan(x,x)) means "for each individual *x*, it is not the case that *x* is taller than *x*" (i.e., nothing is taller than itself). The existential

TABLE G.1 Operator Symbols

Logic Symbol	LogiQL Symbol	Operation Name	English Reading
~	!	Negation	Not; it is not the case that
∧	,	Conjunction	And
∨	;	Inclusive disjunction	Or (inclusive-or)
→	->	Implication	Implies; if … then … ; only if
←	<-	Converse implication	If
∀		Universal quantifier	For all; for each; for every
∃		Existential quantifier	There exists; there is some

quantifier ∃ means "there exists at least one" or "there is some." For example, the following formula may be used to state that "some person is taller than Norma": `∃x(Person(x)∧isTallerThan(x,norma))`.

Table G.1 lists basic correspondences between operator symbols in predicate logic and LogiQL.

Both LogiQL and logic use parentheses '`()`' to group items together, either to list the arguments of a predicate, for example, `likes(x,y)`, or to ensure that a compound expression inside parentheses is evaluated before operating on it from outside. Negation has the highest priority, so it has minimum scope. Hence the following expressions, first in logic and then in LogiQL, mean "*p* is not tall and *p* is not male":

```
~isTall(p) ∧ ~isMale(p)

!isTall(p), !isMale(p)
```

The following expressions use parentheses to conjoin the atoms before negation is applied, and hence mean "it is not the case that *p* is both tall and male":

```
~(isTall(p) ∧ isMale(p))

!(isTall(p), isMale(p))
```

Logic notations typically also allow square brackets '`[]`' as delimiters of expressions to be evaluated before connecting them to outside expressions. For example, the first formula above is equivalent to `~[isTall(p)∧isMale(p)]`. However, LogiQL uses square brackets to delimit the arguments of a function term (e.g., `fatherOf[p1]=p2`).

The logic formulas in this appendix do not assume any precedence among the binary operators. In LogiQL, however, the *and* operator has priority over the *or* operator, which itself has priority over the *implication* operators. Hence, each of the following formulas may be read as "*p* either is male and tall, or is female and short":

```
[isMale(p) ∧ isTall(p)] ∨ [isFemale(p) ∧ isShort(p)]

isMale(p), isTall(p) ; isFemale(p), isShort(p)
```

The semantics of the five propositional operators shown above are provided by the following truth tables, where `p` and `q` denote propositions, and `1` and `0` denote the truth values `True` and `False`, respectively. First, the negation operator reverses the truth value of its argument, so negating a true proposition results in a false proposition, and negating a false proposition results in a true proposition. This is shown in Table G.2.

A conjunction is true if and only if all of its conjuncts are true. An inclusive disjunction is true if and only if at least one of its disjuncts is true. A material implication `p→q` is true unless its antecedent `p` is true and its consequent `q` is false. Hence, `p→q` evaluates to true if `p` is false, and the same is true of `q←p` (Table G.3). Later we discuss some differences between the implication operators of logic and the *arrow* operators of LogiQL.

Over a finite domain, universal quantification is equivalent to the conjunction of its instantiations, and existential quantification is equivalent

TABLE G.2 Truth Table for Negation Operator

p	**~p**
1	0
0	1

TABLE G.3 Truth Table for Binary Logical Operators

p	**q**	**p ∧ q**	**p ∨ q**	**p → q**	**p ← q**
1	1	1	1	1	1
1	0	0	1	0	1
0	1	0	1	1	0
0	0	0	0	1	1

to the disjunction of its instantiations. For example, if the domain of individuals is {a,b,c} and Φ(x) is a FOL formula referring to x, then ∀xΦ(x) is equivalent to Φ(a) ∧ Φ(b) ∧ Φ(c), and ∃xΦ(x) is equivalent to Φ(a) ∨ Φ(b) ∨ Φ(c).

In first-order logic, predicates and quantifiers range over individuals only. In logic, a specific individual may be denoted by an individual literal (e.g., 2) or by a function term (e.g., sqrt(4)). In LogiQL, function terms are indicated with square brackets (e.g., squareRootOf[4]). In logic, individual variables are usually denoted by letters at the end of the alphabet, possibly subscripted (e.g., x, y, z, x_1, x_2). In LogiQL, individual variables typically start with one or more letters, and may include certain other characters such as digits, underscores, and colons (e.g., p, c, person1, car2).

In a classical Datalog rule, head variables are understood to be universally quantified, and variables that occur only in the body are assumed to be existentially quantified. For example, the following predicate logic formula for grandparenthood is equivalent to the LogiQL rule below it. Note that our logic notation allows a quantifier to be followed by a list of individual variables; hence, ∀x,y is shorthand for ∀x∀y.

```
∀x,y[isGrandparentOf(x, y) ← ∃z(isParentOf(x, z) ∧
     isParentOf(z, y))]
```

```
iGrandparentOf(x, y) <- isParentOf(x, z),
     isParentOf(z, y).
```

In LogiQL, the anonymous variable (‘ _ ’) is used as shorthand for an existentially quantified variable that is not used elsewhere in the rule. For example, the following predicate logic formula and LogiQL rule each mean that “*p* is a driver if *p* drives something”:

```
∀p[Driver(p) ← ∃x(drives(p, x))]
```

```
Driver(p) <- drives(p, _).
```

In LogiQL, all predicates are typed, so the types of their arguments are known. For example, each of the following declarations restricts the speaks predicate to range over (person, language) pairs:

```
∀x,y[speaks(x, y) → Person(x) ∧ Language(y)]
```

```
speaks(p, lang) -> Person(p), Language(lang).
```

In LogiQL, if a variable that occurs only in the body of a rule lies in the scope of a *negation* operator, its implicit existential quantifier is assumed to be placed after the *negation* operator. This is the case for both named and anonymous variables. Hence, it might appear that the two following formulas are equivalent, each meaning "*p* is illiterate in Asian languages if *p* is a person who does not speak something that is Asian":

```
∀p[AsianLanguageIlliterate(p) ←
    (Person(p) ∧ ~∃l(speaks(p, l) ∧ isAsian(l)))]

AsianLanguageIlliterate(p) <- Person(p),
  !(speaks(p, l), isAsian(l)).
```

Unfortunately, LogiQL does not support negations in this form. Instead, the same formula can be expressed using two rules as follows:

```
∀p[AsianLanguageIlliterate(p) ←
    (Person(p) ∧ ~∃l(speaks(p, l) ∧ isAsian(l)))]

AsianLanguageLiterate(p) <- speaks(p, l), isAsian(l).
AsianLanguageIlliterate(p) <- Person(p),
    !AsianLanguageLiterate(p).
```

In the above logic formula, the quantification ∀p has scope over the rest of the formula, and the quantification ∃l has scope over the conjunction after it. Each occurrence of the variable *p* is *bound* to the universal quantifier, and each occurrence of the variable *l* is bound to the existential quantifier. Binding multiple occurrences of the same variable to the same quantifier ensures that when the formula is instantiated, each of those occurrences of the variable is replaced by the same instance.

As an example with a negated, anonymous variable, the following formulas each mean that "a non-driver is a person who does not drive anything":

```
∀p[NonDriver(p) ← (Person(p) ∧ ~∃x(drives(p, x))]

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

The following formulas each mean that "*p* is strictly fasting if *p* is a person who does not drink anything and does not eat anything." The first existential quantification ∃x has scope over just `eats(p,x)`. Both occurrences of the `x` variable in `∃x(eats(p,x))` are bound to the first ∃ quantifier. The second existential quantification ∃x has scope over just drinks(p,x). Both occurrences of the x variable in ∃x(drinks(p,x)) are bound to the second ∃ quantifier. The five occurrences of p in the whole formula are bound to the same universal quantifier. Hence, when the formula is instantiated, each p must be replaced by the same item, but the item replacing the x in eats(p,x) may differ from the item replacing the x in drinks(p,x). Similarly, the two anonymous variables in the LogiQL formula are not required to denote the same individual thing:

```
∀p[isStrictlyFastingy(p) ←
    (Person(p) ∧ ~∃x(eats(p, x)) ∧ ~∃x(drinks(p, x)))]
```

```
isStrictlyFasting(p) <- Person(p), !eats(p, _),
    !drinks(p, _).
```

In the context of just the NonDriver rule body (Person(p) ∧~∃x (drives(p,x))), only the variable x is bound. Within that body, the variable p is said to be *free* or *unbound*, even though in the context of the whole rule it is bound to the universal quantifier.

Recall the following safety condition discussed in Unit 3.2.

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.*

Another way of thinking about this safety condition is that variables that are free in the body of the rule must occur in a positive context as the argument of a domain predicate or a domain equality. In the LogiQL formulation of the AsianLanguageIlliterate and NonDriver rules given above, the body variables x and _ occur in a negative context but do not violate SC2 because they are not free in the body. (They are bound to the implicit existential quantifier.) Understanding this may help make sense of error messages from the LogiQL compiler concerning unbound variables in unsafe rules.

In LogiQL, many entity types are declared using a refmode for their reference scheme. For example, the following declaration indicates that Country is an entity type whose instances are referenced by country codes, which are represented by character strings:

```
Country(c), hasCountryCode(c:cc) -> string(cc).
```

This declaration is equivalent to the following set of logical formulas. The last three formulas capture the injective (mandatory, 1:1) nature of the refmode predicate:

$$\forall x[\mathtt{Country}(x) \rightarrow \mathtt{Entity}(x)]$$
$$\forall x,y[\mathtt{hasCountryCode}(x,\ y) \rightarrow (\mathtt{Country}(x) \wedge \mathtt{string}(y))]$$
$$\forall x[\mathtt{Country}(x) \rightarrow y(\mathtt{hasCountryCode}(x,\ y))]$$
$$\forall x,y_1,y_2[(\mathtt{hasCountryCode}(x,\ y_1) \wedge \quad \mathtt{hasCountryCode}(x,\ y_2)) \rightarrow y_1 = y_2]$$
$$\forall x,x_1,y_2[(\mathtt{hasCountryCode}(x_1,\ y) \wedge \quad \mathtt{hasCountryCode}(x_2,\ y)) \rightarrow x_1 = x2]$$

In LogiQL, facts are asserted using delta rules with a '+' modifier. For example, given the refmode declaration above for `Country`, the following code may be used to assert that there is a country that has the country code "AU":

```
+Country("AU").
```

This is actually just convenient shorthand for the following longer assertion, whose expansion can be inferred from the refmode declaration:

```
+Country(c), +hasCountryCode(c:"AU").
```

This is equivalent to the following assertion in logic:

$$\exists x[\mathtt{Country}(x) \wedge \mathtt{hasCountryCode}(x,\ \texttt{"AU"})]$$

Internally, LogiQL identifies entities by the combination of an autogenerated number (e.g., 0, 1, 2, etc.) and their type (e.g., `Country`, `Person`). However, refmodes provide a far more convenient way for humans to identify entities in natural communication.

LogiQL allows entities with refmodes to be referenced simply by their reference values in other contexts as well, since the compiler can always use the refmode declaration to infer the relevant expansion. For example, given the following declaration

```
countryNameOf[c] = cn -> Country(c), string(cn).
```

the following code can be used to add the country name of Australia:

```
+countryNameOf["AU"] = "Australia".
```

This is equivalent to the following assertion in logic:

```
∃x[hasCountryCode(x, "AU") ∧
   hasCountryName(x, "Australia")]
```

Given the refmode declaration above, the assertion "+Country("AU")" is now implied, so there is now no need to explicitly assert it.

In logic, a *sentence* is a well-formed formula with no free variables, and every sentence expresses a proposition, and hence is either true or false. A collection of items is said to be monotonically increasing if over time it either remains constant or has new items added. A system built on classical logic is *monotonic* (more precisely, *monotonic increasing*) because you can add more sentences to it, but you can't retract any sentence. In contrast, LogiQL supports retraction by use of delta predicates with the retraction modifier ('-'). Consider, for example, the following delta rule:

```
-Country("AP").
```

Rather than simply expressing the proposition that there is no country with the country code "AP," this is a command to retract the fact (if any) that there is a country with the country code "AP" (if no such fact is present, no action is taken). Hence, if the previous state of the extensional database (EDB) included the fact `Country("AP")`, execution of the above delta rule would remove this fact from the EDB. Hence, use of retraction rules makes LogiQL *non-monotonic* (i.e., facts may be removed from its EDB). The capability of retraction is often needed in practical information systems, either to remove a fact that is no longer true or of interest, or to correct a mistaken entry.

Although delta rules to simply insert an atom could be viewed as merely asserting propositions, all other delta rules must be viewed as action rules, and if they include the *left-arrow* operator they are either event–action rules, condition-action rules, or event-condition-action rules.

In logic, the formula `p(x)→q(x)` is equivalent to the formula `q(x)←p(x)`. However, in LogiQL, `p(x)->q(x)` is not equivalent to `q(x)<-p(x)`. This is because in LogiQL, right-arrow rules are treated as *constraints* on the EDB, while left-arrow rules are treated as *derivation rules* for inferring new facts. Consider, for example, the following program and data, which are accessible along with sample queries in the files `Pass.logic` and `PassData.logic`:

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

```
genderOf[p] = g -> Person(p), Gender(g).
hasGenderCode(_:gc) -> gc = "M" ; gc = "F".
Person(p) -> genderOf[p] = _.
isIndustrious(p) -> Person(p).
isIntelligent(p) -> Person(p).
passes(p) -> Person(p).
fails(p) -> Person(p).
passes(p) <- isIndustrious(p) ; isIntelligent(p).
fails(p) <- Person(p), !passes(p).

// Data
+genderOf["Adam"] = "M", +isIndustrious("Adam"),
+genderOf["Eve"] = "F", +isIntelligent("Eve").
+genderOf["Bob"] = "M".
```

Since persons and genders have refmodes, facts about them may be entered in the abbreviated form shown, using just their refmode values to identify them. For example, `genderOf["Adam"]="M"` expands internally to the equivalent of the following logic:

```
∃x,y[genderOf(x) = y ∧ hasPersonName(x, "Adam") ∧
    hasGenderCode(y, "M")]
```

The value constraint on `hasGenderCode` of the program ensures that its second argument is populated only with the values "`M`" and "`F`". Any update attempt to use a different gender code will be rejected by the system. Similarly, the constraint on `Person` ensures that any attempt to add persons without their gender will be rejected.

In contrast, the derivation rules at the end of the program are used to draw inferences. If a person is asserted to be either industrious or intelligent, then the system infers that he/she passes. For the data shown, querying the `passes` predicate returns "`Adam`" and "`Eve`". The second derivation rule is used to infer that a person fails if he/she is not known (either by assertion or inference) to pass. For the data shown, querying the `fails` predicate returns `Bob`.

Classical logic adopts the *open-world assumption*, allowing that some facts may simply be unknown. So the absence of a fact does not imply that it is false. However, in logic, you can directly assert that some proposition is false simply by negating it. For example, in logic you could assert that there is no country with the ISO two-letter country code "AP" as follows:

```
~∃x(hasCountryCode(x, "AP"))
```

LogiQL derivation rules with a negation in their body apply the *closed-world assumption*. This approach assumes all relevant facts are known, so the failure to find the fact that Bob passes (either by inspecting the EDB or by inferring new facts from the derivation rules) is interpreted to mean that Bob does not pass. This *negation as failure* semantics differs from classical logic, where the open-world assumption entails that it is unknown whether Bob passes, and consequently, it cannot be inferred that Bob fails.

Thus, LogiQL's inferencing capabilities allow some conclusions to be drawn that do not follow from classical logic. But first-order logic allows many kinds of inferences to be made that are not possible in LogiQL. For example, in propositional logic, the argument of the form $p \rightarrow r$, $q \rightarrow s$, $p \lor q$, and therefore, $r \lor s$ can be trivially shown to be valid using a truth table. However, LogiQL does not allow disjunctions in fact assertions or in the heads of LogiQL rules, so it cannot support disjunctive inferences.

LogiQL also forbids negations in the head of derivation rules and has other restrictions such as the safety conditions considered in Chapter 3. Although these restrictions limit the power of LogiQL, they guarantee that legal programs in LogiQL will always execute in a finite time. In contrast, first-order logic is undecidable, meaning that there are some forms of argument that cannot be evaluated in a finite time.

As discussed in Chapter 4, LogiQL extends classical Datalog by allowing head existentials. Recall the following example, where `presidentOf` is used as a constructor to derive the existence of a country's president from the existence of the country:

```
President(p) ->.
presidentOf[c] = p -> Country(c), President(p).
lang:constructor(`presidentOf).
President(p), presidentOf[c] = p <- Country(c).
```

The final line of code is equivalent to the following logic formulation:

```
∀x[∃y(President(y) ∧ presidentOf(x) = y) ← Country(x)]
```

The existential quantifier in the rule head is implicit in the LogiQL formulation. The rule may be verbalized as "For each country, there is a president who is the president of that country."