Appendix C: LogiQL Syntax

This appendix provides an overview of LogiQL's syntax. Its goal is to give a general picture of how LogiQL programs are structured. Hence, some details have been left out and some nuances glossed over. Moreover, you should be aware that even though a program segment is syntactically valid, executing it still may lead to unexpected results or error messages.

The notation used in this appendix is a version of EBNF (Extended Backus Naur Form) in which syntax categories are separated from their definitions via '`::=`' and are terminated with a period ('`.`'). Literal text is surrounded by quotation mark symbols ('`"`'). Some definitions have alternatives separated by a vertical bar ('|'), and optional constructs are suffixed with a question mark ('?'). Appending an asterisk ('*') to an item indicates zero or more occurrences of that item, and adding a plus sign ('+') indicates one or more occurrences of that item. Finally, parentheses are used to group syntactic elements.

LEXICAL SYNTAX

The lexical syntax of LogiQL is typical of other programming languages. We make special note here only of unusual features:

```
comment ::= "//" text to end of line |
    "/*" text "*/".
identifier ::= (letter | digit | "$" | "?" | "_" |
               ":")+.
```

Identifiers may not begin with digits, and the use of an underscore as the first character of an identifier is interpreted specially by the LogiQL engine. Note also that identifiers may contain embedded colons (':'):

```
integer ::= digit+.
fpnum ::= integer ("." integer)? (exponent)? "f".
```

```
exponent ::= ("e" | "E") ("+" | "-")? integer.
decnum ::= integer (("." integer) | "d" |
                         ("." integer d?)).
string ::= """ text """.
boolean ::= "true" | "false".
```

LogiQL provides a `datetime` literal, delineated with number signs (‘#’):

```
datetime ::= "#" date (time)? (timezone)? "#"
date ::= integer "/" integer "/" integer.
time ::= integer ":" integer (":" integer)?.
timezone ::= text.
```

Strings are contained within paired quotation mark characters (‘`"`’) and may contain *escapes* indicating the occurrence of otherwise unrepresentable characters, such as newlines. Escape sequences take one of two forms, both beginning with a backslash character (‘\’). In the first form, the remainder of the sequence consists of one of the characters (‘`"`’, ‘`b`’, ‘`t`’, ‘`n`’, ‘`f`’, ‘`r`’, ‘\’) standing for, respectively, a quotation mark, a backspace, a tab, a newline, a form feed, a carriage return, or a backslash character. In the second form, the backslash character is followed by a lowercase ‘`u`’ (for *Unicode*) and exactly four hexadecimal characters (`'0-9,A-F,a-f'`) that provide the hexadecimal code for the desired Unicode character.

Braces (‘{’ and ‘}’) may be used for grouping of program elements. In particular, hierarchical assertions and module declarations use braces.

GRAMMAR

A LogiQL `program` consists of a series of `clauses`, which can take several forms:

```
program ::= clause*.
clause ::= fact | rule | constraint.
```

The simplest form of `clause` is called a `fact`, and it consists of a single formula:

```
fact ::= formula.
```

Facts are used to describe the population of the predicate mentioned in the formula.

Slightly more complex are `rules`, which look like the following:

```
rule ::= formula "<-" formula.
```

In the case of `rules`, there are two formulas separated by a left-hand arrow. The intent of a `rule` is to say that if the right-hand formula (the body) evaluates to `true`, then so must the left-hand formula (the head).

The third form of `clause` is called a `constraint` in the LogiQL grammar. `Constraints` are used either to declare predicates or to limit the facts that can populate them. Grammatically, there are three forms of constraints:

```
constraint :: = formula "->" formula.
constraint :: = "->" formula.
constraint :: = "!" formula.
```

In the first form, the intent is to say that if the left-hand side evaluates to `true`, then the right-hand side must also evaluate to `true`. In the second form, there is an implicit `true` on the left-hand side. In the third form the right-hand side has an explicit negation, and there is an implicit `true` on the left-hand side and an implicit arrow. If, during execution, any constraint fails to hold, then that constraint is violated, and execution of the transaction containing the constraint is aborted.

Most `formulas` are compound, being formed from smaller `formulas` using some form of punctuation. For example, comma ('`,`') is used to express a conjunctive formula built up from two or more other formulas. Similarly, semicolon ('`;`') is used in disjunctive formulas. A negation formula is denoted by prepending an exclamation point ('`!`'), and parentheses ('`(`' and '`)`') may be used to group `formulas` for readability or to express precedence.

There are three other kinds of `formulas`. The first involves `atoms`, the second involves aggregation functions, and the third involves `exprs` (expressions), possibly separated by `comparators` (comparison operators):

```
formula ::= atom
          | aggregation
          | expr (comparator expr)+
          | formula "," formula
          | formula ";" formula
          | "!" formula
          | "(" formula ")".
```

An `atom` comprises an `identifier` providing a name for a predicate and a parenthesized `arglist` (argument list), where positions in the list correspond to the predicate's roles. `Arglists` are of two forms. The first is a comma-separated list of `exprs`, while the second comprises a single refmode reference. The `identifier` may optionally be preceded by a `deltaop`, indicating that the predicate population is to be changed. Similarly, an optional stage suffix may be appended to the `identifier`, giving the programmer access to interim execution states of the predicate:

```
atom ::= (deltaop)? identifier (size)? (stage)?
     "(" arglist ")".
deltaop ::= "+" | "-" | "*" | "^".
size ::= "[" integer "]".
stage ::= "@" ("prev" | "previous" | "init" |
          "initial" | "final").
arglist ::= (expr ("," expr)*)?
          | identifier ":" (identifier | constant).
comparator ::= "=" | "!=" | "<" | ">" | "< =" | "> = ".
```

Aggregation functions have their own syntax, looking like the following:

```
aggregation ::= "agg<<" identifier "=" atom ">>"
                formula.
```

The final, major element of LogiQL syntax is the `expr`, typically used to denote a value:

```
expr ::= identifier
        | literal
        | expr arithop expr
        | identifier "[" arglist "]"
        | "(" expr ")".
literal ::= string | boolean | fpnum | decnum |
     integer | datetime.
arithop ::= "+" | "-" | "*" | "/".
```

`Exprs` can take several forms. The simplest `exprs` are either `identifiers` or `literals`. Other `exprs` are used to indicate functional application if they take the form of an `identifier` followed by an `arglist` in square brackets (‘`[`’ and ‘`]`’). Still other `exprs` express arithmetic combinations of simpler `exprs`. Finally, as with `formulas`, `exprs` can be grouped by surrounding them with parentheses.