Appendix D: Built-In Operators and Predicates

This appendix provides a brief summary of many of the most useful operators and predicates that are pre-defined in LogiQL.

BUILT-IN OPERATORS

Logical Operators

The main logical operators are shown in Table D.1. These are also called *propositional operators* because they operate on propositional expressions to form another proposition. The priority indicates the relative order in which the operators are evaluated unless over-ridden by use of parentheses. Negation has highest priority (1), conjunction has precedence over disjunction, and the arrow operators have lowest priority (4). Expressions inside parentheses are evaluated before operating on them from outside.

For example, the following rules are equivalent:

```
AdultPerson(p) <-
    MalePerson(p), !Boy(p) ; FemalePerson(p), !Girl(p).
AdultPerson(p) <-
    (MalePerson(p), !Boy(p)) ; (FemalePerson(p),
    !Girl(p)).
AdultPerson(p) <-
    (MalePerson(p) ; FemalePerson(p)), !(Boy(p) ;
    Girl(p)).
```

TABLE D.1 Logical Operators

Symbol	Meaning	Priority	Operation Name
!	Not	1	Negation
,	And	2	Conjunction
;	Or	3	Disjunction
->	Implies	4	Implication
<-	If	4	Converse implication

Numeric Operators

Numeric operators, also known as *arithmetic operators*, operate on numeric expressions to return a numeric value. Unary negation ('–') has highest priority, and unary addition is not supported. Multiplication ('*') and division ('/') have precedence over addition and subtraction. Operators with the same priority are evaluated left to right. For example, given the following rules, querying `n1` and `n2` returns 4 and 2, respectively. LogiQL numeric operators are shown in Table D.2.

```
n1[] = -1+4/2*3-1.  // n1 evaluates to 4.
n2[] = 8/2/2.       // n2 evaluates to 2.
```

When operating on integer expressions, the *division* operator ('/') performs integer division, removing any fraction from the result. To retain the fraction, use the `float:divide` function instead. For example, given the following rules, if you query the predicate `n3` you will get `1` as the result, and if you query the predicate `n4` you will get `1.16667`:

```
n3[] = 7/(2*3).                  // n3 evaluates to 1.
n4[] = float:divide[7f, 2f*3f].  // n4 evaluates to
                                 // 1.16667.
```

When placed between `string` expressions, the '+' operator performs `string` concatenation. For example, given the following code, querying `name` returns "`Albert Einstein`":

```
name[] = n -> string(n).
name[] = "Albert" + " " + "Einstein".
```

Comparison Operators

The comparison operators shown in Table D.3, also known as *comparators*, can be used to compare the values of scalar expressions

TABLE D.2 Numeric Operators

Symbol	Meaning	Priority
–	Unary minus	1
*	Multiply	2
/	Divide	2
+	Add	3
–	Subtract	3

TABLE D.3 Comparison Operators

Symbol	Meaning
=	Is equal to
! =	Is not equal to
<	Is less than
>	Is greater than
< =	Is less than or equal to
> =	Is greater than or equal to

of primitive types (numbers, `string` and `datetime`). The *ordering comparators* are ('<', '>', '<= ', '>= '). When applied between `strings`, alphabetic ordering is used (e.g., "`Alan`" < "`Ann`"). All comparators have the same priority.

An ordering comparator may be immediately followed by another ordering comparator. For example, the following formula is shorthand for the formula below it:

```
0 < = n < 10
0 < = n, n < 10 // Same as above.
```

If `n` is an integer variable, the formula means that `n` is a digit (i.e., one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

Precedence

Numeric and string operators have precedence over comparators, which in turn have precedence over logical operators. For example, the following two formulas are equivalent:

```
!2 + 3 > 2 * 3
!((2 + 3) > (2 * 3)) // Same as above.
```

Table D.4 summarizes the overall priorities of the main operators discussed.

TABLE D.4 Precedence of Operators

Symbol	Operator Type	Priority
-	Numeric (unary minus)	1
*, /	Numeric (multiply, divide)	2
+, -	Numeric (add, subtract)	3
+	String (concatenate)	3
=, !=, <, >, <=, >=	Comparator	4
!	Logical (negation)	5
,	Logical (conjunction)	6
;	Logical (disjunction)	7

BUILT-IN PREDICATES

Numeric Functions

We now summarize many of the most useful built-in predicates in Table D.5. *Numeric functions* take one or more numeric arguments and return a number. Note the use of the square brackets instead of parentheses around the argument(s).

The next three mathematical functions, shown in Table D.6, are trigonometric. The argument of each function is an angle. In the right-angled triangle shown in Figure D.1, the sine of the angle θ is the ratio of the opposite side (*a*) to the hypotenuse (*h*)—that is, *sine*(θ) = *a*/*h*. The cosine is the ratio of the adjacent side to the hypotenuse—that is, *cosine*(θ) = *b*/*h*. The tangent is the ratio of the opposite side to the adjacent side—that is, *tan*(θ) = *a*/*b*. The angles of a triangle add up to 180 degrees, which equals θ radians. The value of θ is approximately 3.1416. When using these functions in LogiQL, the angle must be expressed in radians rather than degrees.

The three mathematical functions shown in Table D.7 deal with exponentials and logarithms. Like π, the *exponential constant e* is irrational. (It cannot be expressed as the ratio of two integers.) Its value is the limit of $(1 + 1/n)^n$ as *n* approaches infinity and is approximately 2.71828. The function e^x is called the *exponential function*. If $e^n = x$, then *n* is the *natural logarithm* of *x*, or logarithm of *x* to base *e*, which is written as $\log_e(x)$ or simply $\log(x)$. If $10^n = x$, then *n* is the *common logarithm*, or logarithm to base 10, of *x*, and is written as $\log_{10}(x)$.

String Functions

String functions take string expressions as arguments. Recall the use of the `string:like(str,pattern)` function for string pattern matching, where *str* is a string expression and *pattern* is a quoted string that may

TABLE D.5 Numeric Functions

Function	Meaning	Examples
`abs[x]`	Absolute value of *x* (i.e., *x* with its sign removed)	`abs[3f] = 3f` `abs[-3f] = 3f`

`ceil[x]`	Ceiling of *x* (i.e., the smallest floating point integer > = *x*)	`ceil[5.0f] = 5f`
		`ceil[5.1f] = 6f`
		`ceil[5.7f] = 6f`

`floor[x]`	Floor of *x* (i.e., the greatest floating point integer < = *x* [a truncation operation])	`floor[5.0f] = 5f`
		`floor[5.1f] = 5f`
		`floor[5.7f] = 5f`

`pow[x,n]`	*x* to the power *n* (i.e., $x^0 = 1$ $x^1 = x$, $x^2 = x*x$, $x^3 = x*x*x$, etc.)	`pow[4f, 0f] = 1f`
		`pow[4f, 1f] = 4f`
		`pow[4f, 2f] = 16f`
		`pow[4f, 3f] = 64f`

`sqrt[x]`	Non-negative square root of *x* (i.e., the non-negative number that returns *x* when multiplied by itself)	`sqrt[4f] = 2f`
		`sqrt[16f] = 4f`
		`sqrt[5f] = 2.23607f`

TABLE D.6 Trigonometric Functions

Function	Meaning	Examples

`sin[x]`	Sine of angle *x* (where *x* is in radians)	`sin[0f] = 0f`
		`sin[3.1416f/2f] = 1f`
		`sin[3.1416f/4f] = 0.707108f`

`cos[x]`	Cosine of *x* (where *x* is in radians)	`cos[0f] = 1f`
		`cos[3.1416f/2f] = 0f`
		`cos[3.1416f/4f] = 0.707108f`

`tan[x]`	Tangent of *x* (where *x* is in radians)	`tan[0f] = 0f`
		`tan[3.1416f/3f] = 1.73206f`
		`tan[3.1416f/4f] = 1f`

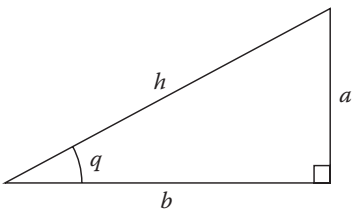

FIGURE D.1 Right triangle used to define trigonometric functions.

TABLE D.7 Exponential and Logarithmic Functions

Function	Meaning	Examples

`exp[x]`	e^x (i.e., *e* to the power *x*)	`exp[0f] = 1f`
		`exp[1f] = 2.71828f`
		`exp[2f] = 7.38906f`

`log[x]`	Power to which *e* must be raised to give x (i.e., *n* where $e^n = x$)	`log[1f] = 0f`
		`log[2.71828f] = 1f`
		`log[10f] = 2.30259f`

`log10[x]`	Power to which 10 must be raised to give *x* (i.e., *n* where $10^n = x$)	`log10[1f] = 0f`
		`log10[2.71828f] = 0.434294f`
		`log10[10f] = 1f`

include an underscore (‘ _ ’), as a wildcard denoting any single character and the percentage character (‘%’), and as a wildcard for any sequence of zero or more characters. For example, `string:like(`*cc*`,"CS%")` is satisfied if `cc` starts with the characters `CS`, and `string:like(`*cc*`,"__ 1%")` is satisfied if `cc` has ‘`1`’ as its third character.

The `string:substring[`*s*`,`*p*`,`*n*`]` function returns the substring of string *s* that starts at position *p*, and is *n* characters in length. Note that the starting position is numbered from 0, so the first character in the source string is at position 0, the second character is at position 1, and so on. For example:

```
string:substring["ABC", 0, 2] returns "AB".
string:substring["ABCDE", 2, 3] returns "CDE".
```

The `string:replace[`*s*`,`*oldPart*`,`*newPart*`]` function replaces each occurrence in string *s* of the substring *oldPart* by *newPart*. For example:

```
string:replace["ACCA", "C", "B"] returns "ABBA".
string:replace["John Smith", "John", "Ann"] returns "Ann
  Smith".
```

The `string` predicates that we have used in this book are summarized in Table D.8.

Type Conversion Functions

The *type conversion* functions `t1:t2:convert[exp]` are used to convert an expression of type *t1* to type *t2*:

```
int:string:convert[65] returns "65".
string:float:convert["65"] returns 65.
```

TABLE D.8 String Functions

Function	Meaning	Examples
`s + t`	Catenation of *s* and *t*	`"ABC" + "DEF" = "ABCDEF"`
`string:length[s]`	Length of string *s*	`string:length[""] = 0` `string:length["abc"] = 3`
`string:like(s,p)`	Match string *s* against pattern *p*	`string:like("ABBCA", "_BB%")`
`string:lower[s]`	Lowercase version of string *s*	`string:lower["ABC"] = "abc"`
`string:replace[s,op,np]`	Replace occurrences of *op* with *np* in *s*	`string:replace["ACCA", "C", "B"] = "ABBA"`
`string:split[s,c,i]`	*i*th segment of *s* as determined by character *c*	`string:split["A B C", " ", 2] = "C"`
`string:substring[s,n,l]`	Substring of *s* starting at position *n* of length *l*	`string:substring["ABCDE", 1, 2] = "BC"`
`string:upper[s]`	Uppercase version of string *s*	`string:upper["abc"] = "ABC"`

Aggregation Functions

Aggregation functions operate on a collection of facts and return a single value for some property of the collection considered as a whole. The four most important of these functions are `count`, `total`, `min`, and `max`. These functions are invoked using the following special syntax, where the variable is assigned the result of evaluating the function over those instances of the collection that satisfy *condition*:

```
agg<<v = count()>> condition
agg<<v = f(x)>> conditionOfx // f is one of total, min, max
```

These four functions are summarized in Table D.9 using examples that have been discussed elsewhere in this book.

In addition to the above aggregation functions, the `seq` and `list` functions can be used to produce sorted results:

```
seq<<v = x>> conditionOfx
```

produces values for `v` in ascending order taken from the values of `x` produced by *conditionOfx*. Alternatively, `list` works with a pair of functions producing the first and next elements produced by an input condition. Uses of `list` take the following form:

```
first(v1), next(v1, v2) <- list<< >> conditionOfv.
```

`first` and `next` can then be used to navigate through the values produced by the condition in ascending order.

TABLE D.9 Aggregation Functions

Function	Meaning and Example

`count()`	Count of the number of instances where the condition is true, e.g.,
	`nrChildrenOf[p] = n <-`
	`agg<<n = count()>> isParentOf(p, _).`

`total(x)`	Sum of the values that satisfy the specified condition, e.g.,
	`totalExpenseOfClaim[c] = t <-`
	`agg<<t = total(e)>> claimItemExpense[c, _] = e.`

`min(x)`	Minimum value of that satisfies the specified condition, e.g.,
	`minIQ[] = n <-`
	`agg<<n = min(iq)>> iqOf[_] = iq.`

`max(x)`	Maximum value of that satisfies the specified condition, e.g.,
	`maxIQof[g] = n <-`
	`agg<<n = max(iq)>> iqOf[p] = iq, genderOf[p] = g.`

Range Predicates

The *range population* predicates *numericType*`:range(`*start*`,`*end*`,`*increment*`,` *x*`)` are useful for populating a variable *x* of *numericType* with a range of numbers from *start* to *end*, incrementing by *increment*. For example, the following populates `rank` with all the integers from 0 through 100:

```
int:range(0, 100, 1, rank)
```

`datetime` Predicates

Several date and time predicates are pre-defined for working with `datetime` values. A date value may be entered as a `#`-delimited string in *mm/dd/yyyy* format. For example, `#02/15/1946#` denotes the date February 15 in the year 1946. Care is needed with `datetime` data, since the time zone defaults to the time zone of the computer used to enter the data. This can easily result in dates being one day off from what you might expect. This issue can be addressed at least partly by including the name of the time zone (e.g., CET for Central European Time) just before the closing `#`. You can also include time data in hours, and optionally minutes and seconds, in *hh*, or *hh*:*mm*, or *hh*:*mm*:*ss* format after the year data. For example, #02/15/1946 15:05:40 GMT# denotes the instant that is 3 hours, 5 minutes, and 40 seconds after midday on February 15, 1946, in the Greenwich Mean Time zone.

The following program and data are used to illustrate some of the more useful `datetime` functions:

```
// Schema
Person(p), hasPersonName(p:n) -> string(n).
birthdateOf[p] = bd -> Person(p), datetime(bd).
deathdateOf[p] = dd -> Person(p), datetime(dd).
birthdateOf[p] = d1, deathdateOf[p] = d2 -> d1 <= d2.

// Data
+birthdateOf["George VI"] = #12/14/1895 GMT#.
+deathdateOf["George VI"] = #02/06/1952 GMT#.
```

The function `datetime:format[`*dt*`,`*formatString*`]` returns the value of the `datetime` variable *dt* in the POSIX date-time format specified in *formatString*. The function `datetime:formatTZ[`*dt*`,` *formatString*`,`*timezone*`]` gives you greater control of the output by including the time zone. For example, the following rule may be used to

reformat the GMT birthdates from the default "`%m/%d/%Y%H:%M:%S`" format to the day-month-year format and to ensure that the time zone remains GMT:

```
dmyGMTbirthdateOf[p] = datetime:formatTZ[bd,
    "%d/%m/%Y", "GMT"] <- birthdateOf[p] = bd.
```

For the sample data, querying the `dmyGMTbirthdateOf` predicate returns

```
George VI, 14/12/1895
```

The function `datetime:part[`*dt*`,`*part*`]` returns the specified part of the `datetime` value. The part is one of "`year`", "`month`", "`day`", "`hour`", "`minute`", "`second`". For example, the following rule may be used to extract just the birth year from the birthdate. For the data shown, querying the `birthYearOf` predicate returns `1895`:

```
birthYearOf[p] = y <-
    birthdateOf[p] = bd, datetime:part[bd, "year"] = y.
```

The function `datetime:partTZ[`*dt*`,`*part*`,`*timezone*`]` extends this function by including the time zone. For example, if, in the above rule, you replace `datetime:part[bd,"year"]` by `datetime:partTZ[bd,"year","GMT"]` you will get the same result, `1895`, for the sample data.

The function `datetime:offset[`*dt1*`,`*dt2*`,`*unit*`]` returns the duration of the interval from *d1* to *d2* measured in terms of the number of specified units. The *unit* is specified as one of "`years`", "`months`", "`days`", "`hours`", "`minutes`", "`seconds`". Care is required when using this offset function, as the value returned is based on simple subtraction. For example, recall the following example from the Chapter 1 Consolidation Exercise. To compute the actual age at death, we need to subtract a year from the approximate death age if the day of year of a person's death occurs before the day of year of the person's birth:

```
approxDeathAgeOf[p] = n <-
    birthdateOf[p] = d1, deathdateOf[p] = d2,
    datetime:offset[d1, d2, "years"] = n.
```

This appendix has now summarized most of the pre-defined operators, functions, and other predicates that are useful in typical applications. LogiQL includes other built-in predicates not discussed here. For a full list of built-in predicates, issue the following command on an existing workspace: `lb workspace list`.