# The AQuA Statement Language

This section defines information about the language used to create AQuA statements. AQuA statement authors will need to be familiar with these requirements, as well as with the relevant AQuA domain ontologies their statement uses.

## Overview and Key Concepts

There are two types of AQuA statements: SELECT statements and EXISTS statements. An EXISTS AQuA statement returns a Boolean value. This allows query authors to test whether certain conditions are met within an enterprise data source as a whole. A SELECT statement returns a list of property values. The SELECT statement is useful when the query author wishes to report back specific information from the enterprise data source whenever certain conditions are met.

Both types of statements have the same main parts:

* Class declarations – The class declarations section names the specific classes from AQuA domain ontologies that are relevant to the AQuA statement. Each named class is assigned a variable that is used to reference instances of the named class later in the AQuA statement. As noted earlier, these class instances are populated from an enterprise’s data source when an AQuA statement is interpreted. More information about the class declarations is provided in section 6.6.1.
* Query body – The query body contains the conditions that are used to examine a set of class instances. Within a query body, properties of a class instance can be extracted, manipulated and compared. Ultimately, the query body conditions produce a value of TRUE (if the conditions are met), FALSE (if the conditions are not met), or UNKNOWN (if information needed to evaluate the conditions is unavailable). More information about the query body is provided in section 6.6.2.
* Return clause – The return clause is only present in a SELECT statement. The return clause enumerates specific class properties that the AQuA statement is to report in its output. More information about the return clause is provided in section 6.6.3

Ultimately, an AQuA statement will take one of the following two forms, depending on whether it is a SELECT statement or an EXISTS statement, respectively:

SELECT ( Class-Declarations ) { Query-Body } Return-Clause

EXISTS ( Class-Declarations ) { Query-Body }

Note that both types of statements use the same rules and constructions for their class declarations and query body portions.

## Whitespaces and Comments

Whitespace is only required in AQuA to the extent that it is needed to separate tokens that would otherwise be merged. (E.g., between the class name and variable name in a class declaration so that the two strings do not run together.) All whitespaces is treated as equivalent and is ignored. AQuA statement authors can and should use whitespace to make their AQuA statements easier for humans to read.

AQuA statements MAY also include comments. These comments follow C++ conventions: all text following a // on a line, and all text between an opening /\* and closing \*/ is ignored when parsing the AQuA statement. The only exception to this is when these character sequences appear within a string literal, in which case they are treated as part of the string literal rather than indicators of a comment. (I.e., the string literal takes precedence during processing.) AQuA statement authors MAY use these comments to clarify the meaning of their AQuA statements.

## Types and Type Conversion

AQuA is a strongly-typed language. In a correctly formatted AQuA statement, every referenced class instance property, comparison, function, and operation has a known type. The data types AQuA uses are described in section 5.1. All functions, comparisons, and operations identify the expected data type of all operands as well as the data type of the result. AQuA statement authors need to ensure that their statements conform to requirements regarding types. AQuA interpreters will not process an AQuA statement that violates data typing rules.

AQuA does not support implicit type conversion; any type conversion must be handled explicitly using type conversions functions. These functions are described in detail in section 6.6.2.7.

## UNKNOWN Values

When an AQuA statement is processed, the AQuA statement maps data from the enterprise data source into the template defined by the AQuA domain ontology class definitions for all classes in the statement’s class declarations. In effect, the interpreter uses a mapping to associate enterprise data to specific named properties, and sets of data to instances of the named classes.

However, this mapping is unlikely to always be complete. Authors of a domain ontology should try to be comprehensive in their description of the entities a given class characterizes, and the data available to an enterprise may not be able to cover the entire breadth of this description. This means that, for any given instance of a class, some of the class’s properties may not be mappable to enterprise data. This would leave the value of these instance properties “unknown”.

To deal with this situation, AQuA employs “three valued logic”. This is an extension of Boolean logic that includes a third possible value, namely “UNKNOWN”, in addition to the traditional TRUE and FALSE values. Similarly, all functions and operations define their behavior when one or more of their inputs are UNKNOWN. As a result, an AQuA statement can be processed correctly even when some instance properties it examines are unknown, although the result of that processing might be that the entire query body result is also UNKNOWN.

UNKNOWN is a universal data value in that any data type can take on a value of UNKNOWN. For any given part of an AQuA statement, the data type of that part is preserved even when the value is unknown. For example, if one had an integer property whose value was UNKNOWN, the property would still have the type of INTEGER and would be usable everywhere an INTEGER type was permissible. In other words, an AQuA statement remains strongly typed regardless of whether any parts of that statement are UNKNOWN at any given time.

## Order of Operations

Within an AQuA statement’s query body, AQuA supports a range of functions, operations, and comparison that can be used to examine property values. A query body can consist of a large number of such parts, provided that they are combined in a manner that preserves AQuA’s requirements for strong typing. As a result, when multiple operations, comparisons, and functions appear together, it is necessary to have a deterministic process for the order in which they are processed.

AQuA largely follows the same order of operations used in the C programming language (with the addition of the power operator, which is not native to C). Specifically, explicit ordering is controlled by the use of parenthesis, while implicit ordering follows the order of operations show in Table 5.

|  |  |  |  |
| --- | --- | --- | --- |
| Precedence | Operator | Description | Associativity |
| 1 | () | Function calls and explicit parentheses | Left-to-right |
| 2 |  |  | Right-to-left |
| + - | Unary plus and minus |
| NOT | Logical NOT |
| 3 | \*\* | Power | Left-to-right |
| 4 | \* / % | Multiplication, division, and modulus |
| 5 | + - | Addition and subtraction |
| 6 | << >> | Bitwise left shift and right shift |
| 7 | < <= | For relational operators < and ≤ respectively |
| > >= | For relational operators > and ≥ respectively |
| 8 | = != | For relational = and ≠ respectively |
| 9 | & | Bitwise AND |
| 10 | ^ | Bitwise XOR (exclusive or) |
| 11 | | | Bitwise OR (inclusive or) |
| 12 | AND | Logical AND |
| 13 | OR | Logical OR |

Table 5: AQuA Order of Operations

Note that care must be taken when converting AQuA statements into other query languages since the implicit order of operations might differ between AQuA and the target query language.

## AQuA Statement Syntax and Semantics

This section provides details regarding the syntax and semantics of AQuA statements. As noted above, there are two types of AQuA statements: EXISTS and SELECT. The former returns a Boolean result while the latter returns a set of values. Each of these verbs results in slightly different AQuA statement structures. A formal description for the two types of statements appear below.

ExistsStatement: EXISTS LPAREN ClassDeclarations RPAREN LEFT\_CURLY QueryBody RIGHT\_CURLY;

SelectStatement: SELECT LPAREN ClassDeclarations RPAREN LEFT\_CURLY QueryBody RIGHT\_CURLY RETURN ReturnClause;

Evaluation of SELECT and EXISTS statements is similar, and only differs in the nature of the final results. In both cases, evaluation of the statement must be equivalent to the following:

1. For each class named in the class declarations, use the enterprise data source to identify instances of this class. An instance corresponds to an occurrence of the entity described by the class as indicated by the enterprise data source. The properties of each instance are populated with the details available in the enterprise data source. Note that some properties will not be mappable to data, and thus would receive a value of UNKNOWN. At the end of step one, there is a set of instances associated with each class variable named in the class declaration.
2. For any evaluation of the query body, each variable is assigned to one of its associated class instances. Evaluate the query body once for every possible combination of these variable-instance associations. In effect, this will result in the query body being run once for every element of the cross produce of the sets of instances identified in step 1.
3. Each evaluation of the query body produces a value of TRUE, FALSE, or UNKNOWN. The final result of the AQuA statement depends on whether the statement is an EXISTS or a SELECT statement. Specifically:

* The result of the evaluation of an EXISTS statement MUST be equivalent to taking the logical OR of the results of evaluating the query body against the enterprise data source.
* The results of the evaluation of a SELECT statement MUST be equivalent to a table containing the values of all fields listed in the return clause for all evaluations of the query body that return TRUE.

Processing rules are described in more detail in section 8.

The class declarations and query body components of both types of AQuA statements are syntactically and semantically identical. In effect, the only difference is how TRUE results of a query body evaluation are handled. As such, apart from the return clause, which only applies to SELECT statements, the syntactic and semantic details provided in the following sections are agnostic as to whether they are used in a SELECT or EXISTS statement.

### Class Declarations

The class declarations part of an AQuA statement is used to identify the set of classes, taken from a set of AQuA domain ontologies, and assign variables to represent instances of those classes. These variables are used within the query body. It is through the declared classes and their associated variables that data from the enterprise data source is made available for evaluation.

The class declarations in an AQuA statement consists of one or more pairs of class name and variable name. The class and variable names are separated by a space, and each class-variable pair is separated by a comma. Each class declarations clause MUST identify at least one class and variable pair, and MAY identify more.

The grammar for a class declarations section appears below:

classDeclarations: classDeclaration (COMMA classDeclaration)\*;

classDeclaration: className variableName;

The className in the above grammar MUST correspond to the name of a class as defined in an AQuA domain ontology. Note that class names are code-specific rather than rendering specific. (See section 5.2.1 for more details.) If a class name cannot be mapped to a class definition, the AQuA interpreter will raise an OntologyNotFound error and terminate processing.

The variableName in the above grammar can be any Unicode string that consists of printable characters and which has no whitespace, with a few exceptions. Specifically, variable names MUST NOT duplicate the names of AQuA language functions and MUST NOT use the name “EXISTS”. Like class names, variables names are code-sensitive, so different character codes are treated as different strings even if they render the same way. (E.g., Unicode characters 0x41 and 0x0391, which both render as "A", would not be treated as equal.)

### Query Body

An AQuA statement’s query body contains conditions against which data from the enterprise data source is tested. These tests use the variables defined in the class declarations part of the statement to reference this data source information via the populated properties of each instance. In effect, the query body contains all of the key logic of an AQuA statement and is where statement authors will spend most of their time.

The query body itself is, in effect, a single Boolean expression in that it evaluates to TRUE, FALSE, or UNKNOWN (using the three-value logic described earlier). The syntax and semantics of a query body are identical to that of a Boolean clause. This Boolean clause can include other typed clauses, per the rules described in the description of Typed Clauses, in section 6.6.2.1.

#### Typed Clauses

As mentioned earlier, AQuA is a strongly typed language. The query body of an AQuA statement can be thought of as a collection of “clauses”, each of which has a type and each of which can be evaluated to a single value of that type. These clauses can be connected and nested within each other so long as doing so preserves the expected types of the respective operators, comparisons, and functions.

There are seven categories of clauses in AQuA:

* Literals – These are statically specified values within an AQuA statement.
* Class Instance Properties – These are named properties of a particular class instance. These represent data extracted from the enterprise data source and exposed through the structure of a domain ontology’s class definition.
* Operators – These are operations on one or two inputs. When there are multiple inputs, both inputs must be of the same type, and the output of the operation is always the same as the type of the inputs.
* Comparisons – These involve comparing two inputs. These inputs MUST have the same type. The result of a comparison always has a type of BOOLEAN.
* Functions – Functions are pre-defined, named processes within the AQuA language. The set of functions is not extensible. All functions specify the required types of their inputs and outputs.
* Type Conversion Functions – These are special instances of functions whose purpose it to take an input of one data type and provide an “equivalent” output in a different data type.
* Inner EXISTS Clause – This clause functions like an EXISTS AQuA statement, but its result is used within another, larger AQuA statement. This allows certain questions about the enterprise data source as a whole (rather than certain combinations of class instances derived from that data source) to factor into the statement’s evaluation. Inner EXISTS clauses always have a BOOLEAN result.

The following sections provide details about each of these types of clauses. Note that the critical element in the use of clauses is the data type of the clause’s result. It is this type that determines where the clause can be used within the overall query body. Clauses are named by the data type of their results. (E.g., a “Boolean clause” will always evaluate to a BOOLEAN type, an “Integer clause” will always evaluate to an INTEGER type, etc.)

#### Literals

Literals represent statically defined values within a query body. Any primitive type can be represented by a literal. AQuA does not support literal class types.

Note that there is no UNKNOWN literal.

The syntax of AQuA literals is described in the following sections.

##### Integer Literals

An integer literal is a sequence of numbers without a decimal point representing a decimal integer. Integer literals MAY be preceded by a unary + or -. Leading 0s in sequence are permitted, but do not impact the value of the literal. (E.g., 00005 and 5 both have the same literal value.) AQuA does not support expressing integer literals in bases other than decimal.

Integer literals MUST be expressed using ASCII-equivalent encoding. Specifically, character codes 48 through 57 represent the numbers 0 through 9, respectively.

An integer literal always has a type of INTEGER, and thus represents one form of an integer clause.

##### Float Literals

A float literal consists of a sequence of numbers that includes exactly one decimal point. This decimal MAY be the final character in the sequence. (This would represent a integer value expressed as a floating point data type.) Float literals MAY be preceded by a unary + or -. Leading 0s in sequence are permitted, but do not impact the value of the literal. (E.g., 00005.3 and 5.3 both have the same literal value.)

Float literals MUST be expressed using ASCII-equivalent encoding. Specifically, character codes 48 through 57 represent the numbers 0 through 9, respectively, and character 46 represents the decimal.

A float literal always has a type of FLOAT, and thus represents one form of a float clause.

##### String Literals

A string literal consists of a sequence of zero or more printable Unicode characters bounded by double quotes. All characters between the double quotes are considered part of the string, including any and all whitespace.

AQuA uses the \ character as the escape designator. Specifically:

* \” represents a double quote within the string literal and the double quote does not terminate the string.
* \\ represents a front-slash within the string literal.
* For all other character pairs preceded by a \ character, the \ is ignored and only the subsequent character is part of the string. (I.e., “\n” is treated as “n”.)

A string literal always has a type of STRING, and thus represents one form of a string clause.

##### Boolean Literal

A Boolean literal is either the value of TRUE or FALSE. This value is not case sensitive. (I.e., true, TRUE, and True all are treated equivalently.) The characters cannot be quoted as “TRUE” is a string literal.

A Boolean literal always has a type of BOOLEAN, and thus represents one form of a Boolean clause.

##### DateTime Literal

A DateTime literal consists of a sequence of characters that conform to the RFC 3339 format of a timestamp. This sequence cannot be quoted as doing so would create a string literal.

A DateTime literal always has a type of DATETIME, and thus represents one form of a DateTime clause.

#### Class Instance Properties

As mentioned above, the variables declared in an AQuA statements class declarations are used within the query body as a way to reference values extracted from the enterprise data source. All such uses of a declared variable are assigned a type in accordance with the associated class’s definition.

A variable can be used directly in the query body as a way to reference the class instance. This is usually done as part of a comparison between two classes. The data type of the variable is always the data type of the specific class whose instance the variable represents.

More commonly, class variables are dereferenced to get individual property values from a class instance. Dereferencing uses a dot notation. For example, consider a class named Software that has a property named Version. Within an AQuA statement’s class declaration, class Software is assigned a variable named S. During evaluation, if one wishes to examine the Version property of instances of class Software, the property instances would be referenced using the syntax S.Version. (I.e., the variable associated with the class of interest – dot – the name of the property of interest.)

Note that some properties within a class are themselves classes. Properties of those sub classes can be dereferenced by continuing the dot notation. For example, consider the classes Software and Computer. The software class has a property named “InstalledOn” whose type is of class Computer. Computer has a property named “Name”. If an AQuA statement’s variable declaration assigned the variable S to instances of the software class, one could write S.InstalledOn.Name. This would provide the value of the Name property of the instance of Computer that populates the InstalledOn property of the instance of Software represented by the variable S. This syntax can be repeated as necessary.

Note that it might be the case where one attempts a chain of property dereferences (e.g., Var.Prop1.Prop2), but an intermediate property has a value of UNKNOWN. When this happens, the value of the whole variable dereference is UNKNOWN. As noted before, this does not change the data type of the statement – in our example above, the whole dereference expression would still have the data type assigned to Prop2 in the relevant domain ontology.

A grammar for specifying the properties of a variable is as follows:

classVarProperty: variableName DOT propertyName (DOT propertyName)\*;

The data type of a class instance’s property is specified in the domain ontology that defines that class. Since a property could be defined to have any type, including a class type, one cannot know the data type of a property without consulting the domain ontology that defines it.

#### Operators

An operator manipulates one or two input operands to produce a particular result value. AQuA defines several operators, but all share the following requirements:

* If an operator takes two operands, all operands MUST be of the same data type.
* The result of an operator is always the same data type as its operands.

The following sections look at the operators defined in AQuA, organized by their data type. Note that there are no string, DateTime, or class type operators defined in AQuA.

##### Integer Operators

All integer operators take one or two integer operands and return a value with a type of INTEGER. As such, all integer operators are forms of integer clauses.

The following integer operators are defined. In the following table, operands A and B always have a data type of INTEGER. If either operand has a value of UNKNOWN, the whole operator evaluates to UNKNOWN.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Syntax** | **Meaning** |
| - (unary) | -A | Negation: Negates the value of A. |
| + | A+B | Addition: Return the sum of A and B. |
| - | A-B | Subtraction: Return B subtracted from A. |
| \* | A\*B | Multiplication: Return the product of A and B. |
| / | A/B | Division: Return A divided by B rounded down to the nearest full integer. Raises a MathException if B is equal to 0. |
| % | A%B | Modulus: Return the remainder of A divided by B. Raises a MathException if B is equal to 0. |
| \*\* | A\*\*B | Power: Returns A raised to the B power. |
| & | A&B | Bitwise AND: Return the bitwise AND of A and B. |
| | | A|B | Bitwise OR: Return the bitwise OR of A and B. |
| ^ | A^B | Bitwise XOR: Return the bitwise exclusive-OR of A and B. |
| >> | A>>B | Right Shift: Shift the bits of A to the right by B places. Bits shifted in from the left are set to 0. |
| << | A<<B | Left Shift: Shift the bits of A to the left by B places. Bits shifted in from the right are set to 0. |

Table 6: Integer Operators

##### Float Operators

All float operators take one or two float operands and return a value with a type of FLOAT. As such, all float operators are forms of float clauses.

The following float operators are defined. In the following table, operands A and B always have a data type of FLOAT. If either operand has a value of UNKNOWN, the whole operator evaluates to UNKNOWN.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Syntax** | **Meaning** |
| - (unary) | -A | Negation: Negates the value of A. |
| + | A+B | Addition: Return the sum of A and B. |
| - | A-B | Subtraction: Return B subtracted from A. |
| \* | A\*B | Multiplication: Return the product of A and B. |
| / | A/B | Division: Return A divided by B. Raises a MathException if B is equal to 0. |
| \*\* | A\*\*B | Power: Returns A raised to the B power. |

Table 7: Float Operators

##### Boolean Operators

All Boolean operators take one or two Boolean operands and return a value with a type of BOOLEAN. As such, all Boolean operators are forms of Boolean clauses.

The following Boolean operators are defined. In the following table, operands A and B always have a data type of BOOLEAN. The respective truth tables use three-valued Kleene logic.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Syntax** | **Meaning** |
| NOT | NOT A | Negation: The logical negation of the Boolean value of A:   |  |  | | --- | --- | | A | NOT A | | FALSE | TRUE | | UNKNOWN | UNKNOWN | | TRUE | FALSE | |
| AND | A AND B | Conjunction: The logical value of the conjunction of A and B:   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | | B | | | | FALSE | UNKNOWN | TRUE | | A | FALSE | FALSE | FALSE | FALSE | | UNKNOWN | FALSE | UNKNOWN | UNKNOWN | | TTRUE | FALSE | UNKNOWN | TRUE | |
| OR | A OR B | Disjunction: The logical value of the disjunction of A and B:   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | | B | | | | FALSE | UNKNOWN | TRUE | | A | FALSE | FALSE | UNKNOWN | TRUE | | UNKNOWN | UNKNOWN | UNKNOWN | TRUE | | TTRUE | TRUE | TRUE | TRUE | |

Table 8: Boolean Operators

Note that Boolean operators are the only operators that might produce a value other than UNKNOWN if one of their operands has a value of UNKNOWN.

#### Comparisons

Comparisons always take two operands and evaluate some comparison between them. AQuA defines several comparisons, but all share the following requirements:

* Both operands of a comparison MUST be of the same type.
* If one or both of the operands of the comparison have a value of UNKNOWN, the comparison evaluates to UNKNOWN.
* All comparisons return a Boolean value and thus are forms of Boolean clauses.

The following sections look at the comparisons defined in AQuA, organized by the data type of their input. Note that there are no comparisons that take Boolean values as inputs.

##### Integer Comparisons

All integer comparators take two integer operands and return a value with a type of BOOLEAN.

The following integer comparisons are defined. In the following table, operands A and B always have a data type of INTEGER. If either operand has a value of UNKNOWN, the whole comparison evaluates to UNKNOWN.

|  |  |  |
| --- | --- | --- |
| **Comparator** | **Syntax** | **Meaning** |
| = | A=B | Equality: True if and only if A and B are numerically equal. |
| != | A!=B | Inequality: True if and only if A and B are not numerically equal. |
| < | A<B | Strictly Less Than: True if and only if A is less than B. |
| <= | A<=B | Less Than or Equal: True if and only if A is less than or equal to B. |
| > | A>B | Strictly Greater Than: True if and only if A is greater than B. |
| >= | A>=B | Greater Than or Equal: True if and only if A is greater than or equal to B. |

Table 9: Integer Comparisons

##### Float Comparisons

All float comparators take two float operands and return a value with a type of BOOLEAN.

The following float comparisons are defined. In the following table, operands A and B always have a data type of FLOAT. If either operand has a value of UNKNOWN, the whole comparison evaluates to UNKNOWN.

|  |  |  |
| --- | --- | --- |
| **Comparator** | **Syntax** | **Meaning** |
| = | A=B | Equality: True if and only if A and B are numerically equal. Precision is not considered. (E.g., 1.4 is equal to 1.400000.) |
| != | A!=B | Inequality: True if and only if A and B are not numerically equal. |
| < | A<B | Strictly Less Than: True if and only if A is less than B. |
| <= | A<=B | Less Than or Equal: True if and only if A is less than or equal to B. |
| > | A>B | Strictly Greater Than: True if and only if A is greater than B. |
| >= | A>=B | Greater Than or Equal: True if and only if A is greater than or equal to B. |

Table 10: Float Comparisons

##### String Comparisons

All string comparators take string operands and return a value with a type of BOOLEAN.

The following string comparisons are defined. In the following table, operands A and B always have a data type of STRING. If either operand has a value of UNKNOWN, the whole comparison evaluates to UNKNOWN.

|  |  |  |
| --- | --- | --- |
| **Comparator** | **Syntax** | **Meaning** |
| = | A=B | Equality: True if and only if A and B are of the same length and have an identical sequence of character codes. Comparison is by the Unicode character code, rather than the rendering. As such, the characters 0x41 and 0x0391 are not equal even though both render as "A". |
| != | A!=B | Inequality: True if and only if A and B are not equal to each other, using the definition of equality given above. |

Table 11: String Comparisons

##### DateTime Comparisons

All DateTime comparators take two DateTime operands and return a value with a type of BOOLEAN.

The following float comparisons are defined. In the following table, operands A and B always have a data type of DATETIME. If either operand has a value of UNKNOWN, the whole comparison evaluates to UNKNOWN. In all cases, if the two DateTime operands are expressed using different time zones, they will be converted to the same time zone before comparison.

|  |  |  |
| --- | --- | --- |
| **Comparator** | **Syntax** | **Meaning** |
| = | A=B | Equality: True if and only if A and B are the same date and time. If fractional seconds are included, the precision of the fractional seconds is not considered so long as the value of the fractional seconds is the same. The capitalization of the “t” and “z” in the timestamps is ignored. |
| != | A!=B | Inequality: True if and only if A and B are not equal according to the rules for equality provided above. |
| < | A<B | Strictly Less Than: True if and only if A temporally precedes B. |
| <= | A<=B | Less Than or Equal: True if and only if A temporally precedes or is equal to B. |
| > | A>B | Strictly Greater Than: True if and only if A temporally follows B. |
| >= | A>=B | Greater Than or Equal: True if and only if A temporally follows or is equal to B. |

Table 12: DateTime Comparisons

##### Class Comparisons

All class comparators take class operands of the same type and return a value with a type of BOOLEAN. Note that it is not sufficient for both operands to be classes – they MUST be the same class type of an AQuA interpreter will throw a TypeError.

The following class comparisons are defined. In the following table, operands A and B always have a data type of the same class. If either operand has a value of UNKNOWN, the whole comparison evaluates to UNKNOWN.

|  |  |  |
| --- | --- | --- |
| **Comparator** | **Syntax** | **Meaning** |
| = | A=B | Equality: True if and only if A and B are both the same instance of the same class. |
| != | A!=B | Inequality: True if and only if A and B are not the same instance of the same class. |

#### Functions

AQuA defines several functions to support manipulation of values within query bodies. The following sections describe the functions defined within AQuA. These sections are organized by the data type of the function’s return value (and thus the type of clause that the function can serve as within a query body). Note that this might differ from the data type of the function’s arguments. AQuA does not define functions that return classes or floating-point values.

In describing these functions, this document uses a shorthand function prototype definition of:

TYPE FunctionName ( Arg1 : TYPE, Arg2: Type ...)

where the TYPE values are replaced with the data type of the function’s return value or arguments, respectively. For example:

STRING SubString( Str : STRING, Start : INTEGER, Length : INTEGER)

describes a function named SubString that takes three arguments (a STRING, INTEGER, and INTEGER, respectively) and returns a value of type STRING.

This section does not discuss function’s whose role it is to convert their operand from one type to another. Those functions are discussed in section 6.6.2.7.

##### Integer Functions

All integer functions return a value of type INTEGER, and thus are forms of integer clauses. For all integer functions, if any arguments have a value of UNKNOWN, the function returns a value of UNKNOWN.

The following integer functions are defined in AQuA.

###### INTEGER INDEXOF ( Str : STRING, Target : STRING)

The function returns the index of the beginning of the first occurrence of the string Target within the string Str. Strings are 0-indexed, so a return value of 0 means that Target appears at the very beginning of Str. The index calculation counts characters, not bytes. As such, a multi-byte Unicode character still only counts as a single character when computing its index. If Target is not found in Str, the function returns -1.

###### INTEGER DAYOFYEAR( D : DATETIME )

Returns the day of year of D as an integer between 1 and 366, inclusive. 1 represents January 1, 365 represents December 31 in a non-leap year, and 366 represents December 31 in a leap year.

###### INTEGER DAYOFMONTH( D: DATETIME )

Returns the day of the month of D as an integer between 1 and 31, inclusive.

###### INTEGER DAYOFWEEK ( D : DATETIME )

Returns the day of the week from D as an integer between 1 and 7, inclusive. 1 represents Sunday while 7 represents Saturday.

###### INTEGER WEEKOFYEAR ( D : DATETIME )

Returns the week of the year from D as an integer between 1 and 52, inclusive.

###### INTEGER SECOND ( D : DATETIME )

Returns the second from D as an integer between 0 and 59, inclusive. Fractional seconds are ignored.

###### INTEGER MINUTE ( D : DATETIME )

Returns the minute from D as an integer between 0 and 59, inclusive. Seconds are ignored.

###### INTEGER HOUR ( D : DATETIME )

Returns the hour from D as an integer between 0 and 23, inclusive. Minutes and seconds are ignored.

###### INTEGER MONTH ( D : DATETIME )

Returns the month from D as an integer between 1 and 12, inclusive.

###### INTEGER YEAR ( D : DATETIME )

Returns the year from D as a 4-digit integer.

###### INTEGER ROUND ( F : FLOAT )

Rounds F down to the nearest whole number and converts it to an integer.

##### String Functions

All string functions return a value of type STRING, and thus are forms of string clauses. For all string functions, if any arguments have a value of UNKNOWN, the function returns a value of UNKNOWN.

The following string functions are defined in AQuA.

###### STRING SUBSTRING ( Str : STRING, Start : INTEGER, Length : INTEGER )

Returns a substring of Str starting with the character at the index of Start and continuing for Length characters. If Start is less than 0, or Length is less than 1, the function MUST raise a runtime ArgumentException. The index calculation counts characters, not bytes. As such, a multi-byte Unicode character still only counts as a single character when computing its index. If Target is not found in Str, the function returns -1.

If the value of Start exceeds the largest index in Str, the function MUST return an empty string. (“”). Otherwise, if there are fewer than Length characters following the Start index, then the function MUST return all characters from the Start index to the end of Str. AQuA statement authors should be aware that this means that the length of the return value of this function might be shorter than expected.

###### STRING REGEX ( Input : STRING, Pattern : STRING )

Return a substring of the Input argument as dictated by the pattern capture described in the Pattern argument. If Pattern is not a valid PCRE regular expression the function MUST raise a runtime ArgumentException. If Pattern does not include at least one capture clause, the function MUST raise a runtime ArgumentException. If Pattern contains more than one capture clause, only the string captured by the first capture clause is returned. If Pattern does not match against Input, then the function returns the empty string (“”).

###### STRING CONCATENATE ( S1 : STRING, S2 : STRING )

Returns the concatenation of S1 and S1.

##### Boolean Functions

All Boolean functions return a value of type BOOLEAN, and thus are forms of boolean clauses.

The following Boolean functions are defined in AQuA.

###### BOOLEAN MATCHES ( Input : STRING, Pattern : STRING )

Returns TRUE if the Input argument is matched by the regular expression in the Pattern argument. If Pattern is not a valid PCRE regular expression the function MUST raise a runtime ArgumentException. If either of the arguments are UNKNOWN, the function MUST return UNKNOWN. If both arguments are known but Pattern does not match against Input, the function returns FALSE.

###### BOOLEAN IsUnknown ( Input : ANY )

IsUnknown is unique among AQuA functions in that its argument can be of any data type. The function returns TRUE if and only if Input is assigned a value of UNKNOWN. It returns FALSE otherwise. The function never returns UNKNOWN.

##### DateTime Functions

The one defined DateTime function returns a value of type DATETIME, and thus is a form of string clause.

The following DateTime function is defined in AQuA.

###### DATETIME CURRENTDATETIME ( )

The function takes no arguments and returns the current date and time at the time this part of the query body is executed. If the current date and time is not available from the system clock the function must raise a fatal RuntimeError. (This is a fatal error because it is likely that it would be raised on every iteration of the query body, and thus it makes little sense to continue.)

Interpreters MAY choose to cache result of the first use of this function when an AQuA statement is evaluated and use that value for all subsequent evaluations of CURRENTDATETIME when processing that statement. This can significantly improve efficiency during evaluation.

#### Type Conversion Functions

Type conversion functions exist to convert an input into an “equivalent” value that is of a different data type. The following sections describe the functions defined within AQuA. These sections are organized by the data type of the function’s return value (and thus the type of clause that the function can serve as within a query body). AQuA does not define any functions that convert between or to class types.

In all cases, if the input value is UNKNOWN then the return value of the function is UNKNOWN. However, this UNKNOWN value would be expressed in the new type. In other words, while the value might not be known, type conversion functions always return values of the new data type.

##### Integer Conversion Functions

The following functions convert an input value into an equivalent integer value. Note that because conversion from float to integer is not lossless, the ROUND function is described in section 6.6.2.6.1.11 under regular integer functions.

###### INTEGER StringToInteger ( S : STRING )

Converts S into an equivalent integer. If S does not match the production rules for an integer literal (section 6.6.2.2.1), then the function MUST raise a TypeConversionException.

###### INTEGER BooleanToInteger ( B : BOOLEAN )

If B is TRUE, returns 1; otherwise, returns 0.

###### INTEGER DateTimeToInteger ( D : DATETIME )

Converts D to an integer representing the equivalent UNIX time. If D is outside of the range representable by a 32-bit UNIX time, the function MUST raise a TypeConversionException.

##### Float Conversion Functions

The following functions convert an input value into an equivalent float value.

###### FLOAT IntegerToFloat ( N : INTEGER )

Convert N into an equivalent floating-point value. Note that, especially for very large integer, the limits of floating point precision might introduce a small amount of error in this conversion

###### FLOAT StringToFloat ( S : STRING )

Convert S into an equivalent floating-point number. If S does not match the production rules for a float literal (section 6.6.2.2.2), then this function MUST raise a TypeConversionException.

##### String Conversion Functions

The following functions convert an input value into an equivalent string.

###### STRING IntegerToString ( N : INTEGER )

Convert N into a string of its decimal value. The output will conform to the production rules for an integer literal. (Section 6.6.2.2.1)

###### STRING FloatToString ( F : FLOAT )

Convert F into a string of its decimal value. The output will conform to the production rules for a float literal. (Section 6.6.2.2.2)

###### STRING BooleanToString ( B : BOOLEAN )

Convert B into “TRUE” if B is TRUE, or “FALSE” if B is FALSE.

###### STRING DateTimeToString ( D : DATETIME )

Convert D into an RFC 3339 conformant string of the same time. This output will conform to the production rules for a DateTime literal. (Section 6.6.2.2.5) The output MUST be expressed using the same time zone as D.

##### Boolean Conversion Functions

The following functions convert an input value into an equivalent Boolean value.

###### BOOLEAN IntegerToBoolean ( N : INTEGER )

If N is not 0, returns TRUE. Otherwise, returns FALSE.

###### BOOLEAN FloatToBoolean ( F : FLOAT )

If F is not 0.0, returns TRUE. Otherwise, returns FALSE.

###### BOOLEAN StringToBoolean ( S : STRING )

If S is not empty, returns TRUE. Otherwise, returns FALSE. Note that this means that StringToBoolean( BooleanToString( FALSE ) ) will return TRUE, since BooleanToString does not return the empty string for a FALSE input.

##### DateTime Conversion Functions

The following functions convert an input value into an equivalent DateTime value.

###### DATETIME IntegerToDateTime ( N : INTEGER )

Converts N into a datetime as if N were a UNIX time value. If N is not a value UNIX time (e.g., if it were larger than the largest possible 32-bit integer) then this function MUST raise a TypeConversionException.

###### DATETIME StringToDateTime ( S : STRING )

Convert S into an equivalent datetime. If S is not a valid production for a DateTime literal (section 6.6.2.2.5) this function MUST raise a TypeConversionException.

#### Inner EXISTS Clause

An inner EXISTS clause behaves in the same way as a regular EXISTS clause, but does so within the context of a surrounding AQuA statement. This can be a useful way to incorporate characteristics of the enterprise data source as a whole within the query.

Inner EXISTS clauses are also sometimes made necessary by the structure of the domain ontologies. Consider a domain ontology that defines COMPUTER and SOFTWARE classes. The SOFTWARE class includes an InstalledOn property that points to an instance of the computer class as well as a Name property that identifies the software in question. Assume the AQuA statement author wishes to check whether there are any computers in the enterprise where the software Named X is present but the software Named Y is not. One might think the following AQuA statement would suffice:

SELECT ( COMPUTER c, SOFTWARE s1, SOFTWARE s2 ) {

s1.Name = “X”

AND s2.Name = “Y”

AND s1.InstalledOn = c

AND NOT s2.InstalledOn = c

} RETURN ....

However, this query will match all iterations where s1 is installed on c and s2 is installed on a computer other than c. This would not result in the information the statement author is looking for.

Instead, an inner EXISTS clause can be used:

SELECT ( COMPUTER c, SOFTWARE s1 ) {

s1.Name = “X”

AND s1.InstalledOn = c

AND NOT EXISTS ( SOFTWARE s2 ) {

s2.Name = “Y”

AND s2.InstalledOn = c

}

} RETURN ...

This second query would match all cases where there was a computer on which X was installed, and for which there was no software named Y installed on that computer.

An inner EXISTS clause follows the same syntax and semantics of a regular EXISTS statement as described in section 6.6. The only additional constraint is that the inner EXISTS clause MUST NOT duplicate a variable name used in any enclosing statement. Evaluation of an inner EXISTS clause is also the same as for regular EXISTS statements.

Because an inner EXISTS clause can be processor intensive to compute, especially when evaluated against large data sources, AQuA statement authors are advised to use inner EXISTS clauses sparingly, and place them in locations where they will only be run if they are needed to determine the truth of a query body. (For example, at the end of a string of ANDed clauses, where the EXISTS clause would be irrelevant if any preceding clause evaluated to FALSE.)

### Return Clause

The return clause specifies the elements to be returned when executing a SELECT statement.

A return is composed of the ‘RETURN’ keyword followed by a comma-separated list of class properties. Class properties are identified using the dot notation used for class instance properties, as described in section 6.6.2.3. All listed properties MUST have a primitive data type; it is a SyntaxError for any of the listed properties to have a class type.

The grammar of the return clause is as follows:

ReturnClause: classVarProperty (COMMA classVarProperty)\*;

The return clause identifies the specific properties to include in the output for all instance combinations where the query body evaluates to TRUE. This is described in more detail in section 8.

## AQuA

grammar AQuA;

aqua: (aquaStatement) EOF;

aquaStatement: (selectStatement | existsStatement);

selectStatement: SELECT LPAREN classDeclarations RPAREN LEFT\_CURLY queryBody RIGHT\_CURLY RETURN returnClause;

existsStatement: EXISTS LPAREN classDeclarations RPAREN LEFT\_CURLY queryBody RIGHT\_CURLY;

returnClause: classVarProperty (COMMA classVarProperty)\*;

queryBody: (NOT)? (nestedStatement | integerStatement | floatStatement | booleanStatement | stringStatement | dateTimeStatement | objectStatement | existsStatement) (booleanJoin queryBody)?;

nestedStatement: LPAREN queryBody RPAREN;

integerStatement: integerOperand (arithmeticOperator | bitwiseOperator | comparisonOperator) integerOperand;

floatStatement: (floatOperand | integerOperand) (arithmeticOperator | comparisonOperator) (floatOperand | integerOperand);

stringStatement: stringOperand equalityOperator stringOperand;

booleanStatement: booleanOperand ((equalityOperator | bitwiseOperator) booleanOperand)?;

dateTimeStatement: dateTimeOperand comparisonOperator dateTimeOperand;

objectStatement: objectOperand equalityOperator objectOperand;

operand: (integerOperand | floatOperand | stringOperand | booleanOperand | dateTimeOperand | objectOperand);

integerOperand: (nestedIntegerStatement | classVarProperty | VariableName | integerFunction | integerType);

floatOperand: (nestedFloatStatement | classVarProperty | VariableName | floatFunction | floatType);

stringOperand: (nestedStringStatement | classVarProperty | VariableName | stringFunction | LiteralValue);

booleanOperand: (nestedBooleanStatement | classVarProperty | VariableName | booleanFunction | booleanType);

dateTimeOperand: (nestedDateTimeStatement | classVarProperty | VariableName | dateTimeFunction | dateTimeType);

objectOperand: (nestedObjectStatement | classVarProperty | VariableName);

nestedIntegerStatement: LPAREN integerStatement RPAREN;

nestedFloatStatement: LPAREN floatStatement RPAREN;

nestedStringStatement: LPAREN stringStatement RPAREN;

nestedBooleanStatement: LPAREN booleanStatement RPAREN;

nestedDateTimeStatement: LPAREN dateTimeStatement RPAREN;

nestedObjectStatement: LPAREN objectStatement RPAREN;

booleanJoin: AND | OR;

arithmeticOperator: PLUS | MINUS | MULTIPLY | DIVIDE | MODULUS | POWER ;

comparisonOperator: EQ | NEQ | LT | LE | GT | GE;

bitwiseOperator: BITAND | BITOR | BITXOR | BSL | BSR;

equalityOperator: EQ | NEQ;

//////////////////////////////

// classes

classDeclarations: classDeclaration (COMMA classDeclaration)\*;

classDeclaration: ClassName VariableName;

classVarProperty: ClassName DOT PropertyName (DOT PropertyName)\*;

//////////////////////////////

// Function structure

integerFunction: integerFunctionNames LPAREN functionArgs RPAREN;

floatFunction: floatFunctionNames LPAREN functionArgs RPAREN;

stringFunction: stringFunctionNames LPAREN functionArgs RPAREN;

booleanFunction: booleanFunctionNames LPAREN functionArgs RPAREN;

dateTimeFunction: dateTimeFunctionNames LPAREN functionArgs RPAREN;

functionArgs: operand (COMMA operand)\*;

//////////////////////////////

// Functions Names

integerFunctionNames: INDEXOFFN | DAYOFYEARFN | DAYOFMONTHFN | DAYOFWEEKFN | WEEKOFYEARFN | SECONDFN | MINUTEFN | HOURFN | MONTHFN | YEARFN | ROUNDFN | STRINGTOINTEGERFN | BOOLEANTOINTEGERFN | DATETIMETOINTEGERFN;

floatFunctionNames: INTEGERTOFLOATFN | STRINGTOFLOATFN;

stringFunctionNames: SUBSTRINGFN | REGEXFN | CONCATENATEFN | INTEGERTOSTRINGFN | FLOATTOSTRINGFN | BOOLEANTOSTRINGFN | DATETIMETOSTRINGFN ;

booleanFunctionNames: MATCHESFN | ISUNKNOWNFN | INTEGERTOBOOLEANFN | FLOATTOBOOLEANFN | STRINGTOBOOLEANFN ;

dateTimeFunctionNames: CURRENTDATETIMEFN | INTEGERTODATETIMEFN | STRINGTODATETIMEFN ;

//Returns Integer

INDEXOFFN: 'INDEXOF';

DAYOFYEARFN: 'DAYOFYEAR';

DAYOFMONTHFN: 'DAYOFMONTH';

DAYOFWEEKFN: 'DAYOFWEEK';

WEEKOFYEARFN: 'WEEKOFYEAR';

SECONDFN: 'SECOND';

MINUTEFN: 'MINUTE';

HOURFN: 'HOUR';

MONTHFN: 'MONTH';

YEARFN: 'YEAR';

//Casting to integer

ROUNDFN: 'ROUND';

STRINGTOINTEGERFN: 'StringToInteger';

BOOLEANTOINTEGERFN: 'BooleanToInteger';

DATETIMETOINTEGERFN: 'DateTimeToInteger';

//Casting to float

INTEGERTOFLOATFN: 'IntegerToFloat';

STRINGTOFLOATFN: 'StringToFloat';

//Returns string

SUBSTRINGFN: 'SUBSTRING';

REGEXFN: 'REGEX';

CONCATENATEFN: 'CONCATENATE';

//Casting to string

INTEGERTOSTRINGFN: 'IntegerToString';

FLOATTOSTRINGFN: 'FloatToString';

BOOLEANTOSTRINGFN: 'BooleanToString';

DATETIMETOSTRINGFN: 'DateTimeToString';

//Returns Boolean

MATCHESFN: 'MATCHES';

ISUNKNOWNFN: 'IsUnknown';

//Casting to Boolean

INTEGERTOBOOLEANFN: 'IntegerToBoolean';

FLOATTOBOOLEANFN: 'FloatToBoolean';

STRINGTOBOOLEANFN: 'StringToBoolean';

//Returns DateTime

CURRENTDATETIMEFN: 'CURRENTDATETIME';

//Casting to DateTime

INTEGERTODATETIMEFN: 'IntegerToDateTime';

STRINGTODATETIMEFN: 'StringToDateTime';

//////////////////////////////

// datatypes

integerType: (PLUS | MINUS)? (Digit | Digits );

floatType: ((PLUS | MINUS)? (Digits DOT Digits | DOT Digits));

Digits: Digit+;

Digit: [0-9];

TRUE: ('TRUE' | 'True' | 'true' | '1');

FALSE: ('FALSE' | 'False' | 'false' | '0');

booleanType: (TRUE | FALSE);

dateTimeType: Date Time;

Date: Year HYPHEN Month HYPHEN Day DATE\_SUFFIX;

Time: Hours COLON Minutes COLON Seconds TIMEZONE\_SUFFIX?;

fragment Year: [1-2] [0-1] [0-9] [0-9];

fragment Month: [0] [1-9] | [1] [0-2];

fragment Day: ([0] [1-9]) | ([1-2] [0-9]) | ([3] [0-1]);

fragment Hours: ([0-1] [0-9]) | ([2] [1-3]);

fragment Seconds: [0-5] [0-9] (DOT [0-9]+)?;

fragment Minutes: [0-5] [0-9];

//////////////////////////////

// Reserved

SELECT: 'SELECT';

EXISTS: 'EXISTS';

RETURN: 'RETURN';

RIGHT\_CURLY: '}' ;

LEFT\_CURLY: '{' ;

//////////////////////////////

// Operators

AND: 'AND';

OR: 'OR';

NOT: 'NOT';

RPAREN : ')' ;

LPAREN : '(' ;

PLUS: '+' ;

MINUS: '-' ;

HYPHEN: MINUS;

DIVIDE: '/' ;

MULTIPLY: '\*' ;

POWER: '\*\*' ;

MODULUS: '%' ;

BITAND: '&';

BITOR: '|';

BITXOR: '^' ;

BSL: '<<' ;

BSR: '>>' ;

EQ: '=';

NEQ: '!=';

LT: '<';

LE: '<=';

GT: '>';

GE: '>=';

//////////////////////////////

// Literals

DOT: '.';

COMMA: ',';

QUOTE: DOUBLEQUOTE;

DOUBLEQUOTE: '\"';

COLON: ':';

DATE\_SUFFIX: 'T';

TIMEZONE\_SUFFIX: 'Z';

//////////////////////////////

// Generic values

LiteralValue: QUOTE (~["])\* QUOTE;

ClassName: [A-Za-z0-9]+;

VariableName: [A-Za-z0-9]+;

PropertyName: [A-Za-z0-9]+;

//////////////////////////////

// Whitespace and comments

WS: [ \n\r\t]+ -> skip;

COMMENT: '/\*' .\*? '\*/' -> skip;

LINE\_COMMENT: '//' ~[\r\n]\* -> skip;