Chapter 1. Basic Operations

Table of Contents

Now that we have showed how expressions can be composed, we can begin the tour of all JSONiq expressions. First, we introduce the most basic operations.

Construction of Sequences

Comma Operator

The comma allows you to concatenate two sequences, or even single items. This operator has the lowest precedence of all, so do not forget the parentheses if you would like to change this.

Also, the comma operator is associative -- in particular, sequences do not nest. You need to use arrays in order to nest.

Example 1.1. Comma

```
1, 2, 3, 4, 5,
{ "foo" : "bar" }, [ 1  ],
1 + 1, 2 + 2,
(1, 2, (3, 4), 5)
```

```
Result:
 1
 2
 3
 4
 5
 {
   "foo" : "bar"
 }
 [ 1 ]
 2
 4
 1
 2
 3
```

```
 4
 5
```

Range Operator

With the binary operator "to", you can generate larger sequences with just two integer operands.

If the left operand is greater than the right operand, an empty sequence is returned.

If an operand evaluates to something else than a single integer, an error is raised. There is one exception with the empty sequence, which behaves in a particular way for most operations (see below).

Example 1.2. Range operator

```
1 to 10,
10 to 1
```

```
Result:
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
```

Parenthesized Expressions

Expressions take precedence on one another. For example, addition has a higher precedence than the comma. Parentheses allow you to change precedence.

If the parentheses are empty, the empty sequence is produced.

Example 1.3. Empty sequence

```
( 2 + 3 ) * 5,
()
```

```
Result:
 25
```

Arithmetics

JSONiq supports the basic four operations, as well integer division and modulo. You should keep in mind that, as is the case in most programming languages, multiplicative operations have precedence over additive operations. Parentheses can override it, as explained above.

Example 1.4. Basic arithmetic operations with precedence override

```
1 * ( 2 + 3 ) + 7 idiv 2 - (-8) mod 2
```

```
Result:
 8
```

Dates, times and durations are also supported in a natural way.

Example 1.5. Using basic operations with dates.

```
date("2013-05-01") - date("2013-04-02")
```

```
Result:
 "P29D"
```

If any of the operands is a sequence of more than one item, an error is raised.

If any of the operands is not a number, a date, a time or a duration, or if the operands are not compatible (say a number and a time), an error is raised.

Do not worry if the two operands do not have the same number type, JSONiq will do the adequate conversions.

Example 1.6. Basic arithmetic operations with different, but compatible number types

```
2.3e4 + 5
```

```
Result:
 23005
```

String Concatenation

Two strings or more can be concatenated using the concatenation operator. An empty sequence is treated like an empty string.

Example 1.7. String concatenation

```
"Captain" || " " || "Kirk",
"Captain" || () || "Kirk"
```

```
Result:
 "Captain Kirk"
 "CaptainKirk"
```

Comparison

Atomics can be compared with the usual six comparison operators (equality, non-equality, lower-than, greater-than, lower-or-equal, greater-or-equal), and with the same two-letter symbols as in MongoDB.

Comparison is only possible between two compatible types, otherwise, an error is raised.

Example 1.8. Equality comparison

```
1 + 1 eq 2,
1 lt 2
```

```
Result:
 true
 true
```

null can be compared for equality or inequality to anything - it is only equal to itself so that false is returned when comparing if for equality with any non-null atomic. True is returned when comparing it with non-equality with any non-null atomic.

Example 1.9. Equality and non-equality comparison with null

```
1 eq null,
"foo" ne null,
null eq null
```

```
Result:
 false
 true
 true
```

For ordering operators (lt, le, gt, ge), null is considered the smallest possible value (like in JavaScript).

Example 1.10. Ordering comparison with null

```
null lt 1
```

```
Result:
 true
```

Comparisons and logic operators are fundamental for a query language and for the implementation of a query processor as they impact query optimization greatly. The current comparison semantics for them is carefully chosen to have the right characteristics as to enable optimization.

The Empty Sequence in Range, Arithmetics and Comparison Operations

In range operations, arithmetics and comparisons, if an operand is the empty sequence, then the result is the empty sequence as well.

Example 1.11. The empty sequence used in basic operations

```
() to 10,
1 to (),
1 + (),
() eq 1,
() ge 10
```

```
Result:
```

Logic

JSONiq logics support is based on two-valued logics: there is just true and false and nothing else.

Non-boolean operands get automatically converted to either true or false, or an error is raised. The boolean() function performs a manual conversion. The rules for conversion were designed in such a way that it feels "natural". Here they are:

- An empty sequence is converted to false.
- A singleton sequence of one null is converted to false.
- A singleton sequence of one string is converted to true except the empty string which is converted to false.
- A singleton sequence of one number is converted to true except zero or NaN which are converted to false.
- Operand singleton sequences of any other item cannot be converted and an error is raised.
- Operand sequences of more than one item cannot be converted and an error is raised.

Example 1.12. Conversion to booleans

```
{
  "empty-sequence" : boolean(()),
  "null" : boolean(null),
  "non-empty-string" : boolean("foo"),
  "empty-string" : boolean(""),
  "zero" : boolean(0),
  "not-zero" : boolean(1e42)
},
null and "foo"
```

```
Result:
 {
   "empty-sequence" : false,
   "null" : false,
   "non-empty-string" : true,
   "empty-string" : false,
   "zero" : false,
   "not-zero" : true
 }
 false
```

Propositional Logic

JSONiq supports the most famous three boolean operations: conjunction, disjunction, and negation. Negation has the highest precedence, then conjunction, then disjunction. Comparisons have a higher precedence than all logical operations. Parentheses can override.

Example 1.13. Logics with booleans

```
  true and ( true or not true ),
  1 + 1 eq 2 or not 1 + 1 eq 3
```

```
Result:
 true
 true
```

A sequence with more than one item, or singleton objects and arrays cannot be converted to a boolean. An error is raised if it is attempted.

Unlike in C++ or Java, you cannot rely on the order of evaluation of the operands of a boolean operation. The following query may return true or may return an error.

Example 1.14. Non-determinism in presence of errors.

```
  true or (1 div 0)
```

```
Result:
 true
```

First-Order Logic (Quantified Variables)

Given a sequence, it is possible to perform universal or existential quantification on a predicate.

Example 1.15. Universal quantifier

```
  every $i in 1 to 10 satisfies $i gt 0,
  some $i in -5 to 5, $j in 1 to 10 satisfies $i eq $j
```

```
Result:
 true
 true
```

Variables can be annotated with a type. If no type is specified, item* is assumed. If the type does not match, an error is raised.

Example 1.16. Existential quantifier with type checking

```
  some $i as integer in -5 to 5, $j as integer in 1 to 10 satisfies $i eq $j
```

```
Result:
 true
```

Builtin Functions

The syntax for function calls is similar to many other languages.

Like in C++ (namespaces) or Java (packages, classes), functions live in namespaces that are URIs.

Although it is possible to fully write the name of a function, namespace included, it can be cumbersome. Hence, for convenience, a namespace can be associated with a prefix that acts as a shortcut.

JSONiq supports three sorts of functions:

- Builtin functions: these have no prefix and can be called without any import.
- Local functions: they are defined in the prolog, to be used in the main query. They have the prefix *local:*. Chapter ??? describes how to define your own local functions.
- Imported functions: they are defined in a library module. They have the prefix corresponding to the alias to which the imported module has been bound to. Chapter ??? describes how to define your own modules.

For now, we only introduce how to call builtin functions -- these are the simplest, since they do not need any prefix or explicit namespace.

Example 1.17. A builtin function call.

```
keys({ "foo" : "bar", "bar" : "foo" }),
concat("foo", "bar")
```

```
Result:
 "foo"
 "bar"
 "foobar"
```

Some builtin functions perform aggregation and are particularly convenient:

Example 1.18. A builtin function call.

```
sum(1 to 100),
avg(1 to 100),
count( (1 to 100)[ $$ mod 5 eq 0 ] )
```

```
Result:
 5050
 50.5
 20
```

Remember that JSONiq is a strongly typed language. Functions have signatures, for example sum() expects a sequence of numbers. An error is raised if the actual types do not match the expected types.

Also, calling a function with two parameters is different from calling a function with one parameter that is a sequence with two items. For the latter, extra parentheses must be added to make sure that the sequence is taken as a single parameter.

Example 1.19. Calling a function with a sequence.

```
count((1, 2, 3, 4))
```

```
Result:
 4
```