JSONiq - the SQL of NoSQL 1.0

JSONiq

The complete reference

Jonathan Robie

Ghislain Fourny

Matthias Brantner

Daniela Florescu

Till Westmann

Markos Zaharioudakis

JSONiq - the SQL of NoSQL 1.0 JSONiq
The complete reference
Edition 1.0.11

Author	Jonathan Robie	*jonathan.robie@gmail.com*
Author	Ghislain Fourny	*ghislain.fourny@28msec.com*
Author	Matthias Brantner	*matthias.brantner@28msec.com*
Author	Daniela Florescu	*dana.florescu@oracle.com*
Author	Till Westmann	*till.westmann@28msec.com*
Author	Markos Zaharioudakis	*markos.zaharioudakis@oracle.com*
Editor	Ghislain Fourny	*ghislain.fourny@28msec.com*

This document provides a complete reference of the core of JSONiq, the JSON query language.

1. Introduction **1**
1.1. Why JSONiq? ... 1
1.2. Structure of a JSONiq program. ... 2

2. JSON and the JSONiq data model **3**
2.1. JSON ... 3
2.2. JSONiq values ... 3
2.3. Objects ... 4
2.4. Arrays ... 4
2.5. Atomics ... 4

3. Collections **5**

4. JSONiq types **7**
4.1. Item types ... 7
4.1.1. Atomics ... 7
4.1.2. JSON items ... 9
4.1.3. The most general item type. ... 9
4.2. Sequence types ... 9

5. Expressions **11**
5.1. Construction of items ... 11
5.1.1. Atomic literals ... 11
5.1.2. Object constructors ... 14
5.1.3. Array constructors ... 16
5.1.4. Composing constructors ... 17
5.2. Basic operations ... 19
5.2.1. Construction of sequences ... 19
5.2.2. Parenthesized expression ... 21
5.2.3. Arithmetics ... 21
5.2.4. String concatenation ... 23
5.2.5. Comparison ... 24
5.2.6. Logics ... 26
5.3. Function Calls ... 29
5.4. Selectors ... 30
5.4.1. Object field selector ... 31
5.4.2. Array member selector ... 33
5.4.3. Sequence predicates ... 35
5.5. Control flow expressions ... 36
5.5.1. Conditional expressions ... 36
5.5.2. Switch expressions ... 38
5.5.3. Try-catch expressions ... 39
5.6. FLWOR expressions ... 40
5.6.1. For clauses ... 41
5.6.2. Where clauses ... 43
5.6.3. Order clauses ... 44
5.6.4. Group clauses ... 46
5.6.5. Let clauses ... 47
5.6.6. Count clauses ... 48
5.6.7. Map operator ... 49
5.6.8. Composing FLWOR expressions ... 49
5.6.9. Ordered and Unordered expressions ... 50
5.7. Expressions dealing with types ... 51
5.7.1. Instance-of expressions ... 51
5.7.2. Treat expressions ... 52
5.7.3. Castable expressions ... 54

5.7.4. Cast expressions 55
5.7.5. Typeswitch expressions 57

6. Prologs **59**
6.1. Setters. 59
6.1.1. Default collation 59
6.1.2. Default ordering mode 60
6.1.3. Default ordering behaviour for empty sequences 60
6.1.4. Default decimal format 60
6.2. Global variables 61
6.3. Functions 63

7. Modules **65**

8. Function Library **67**
8.1. JSON specific functions. 67
8.1.1. keys 67
8.1.2. members 67
8.1.3. null 68
8.1.4. parse-json 68
8.1.5. size 68
8.1.6. accumulate 69
8.1.7. descendant-arrays 69
8.1.8. descendant-objects 69
8.1.9. descendant-pairs 69
8.1.10. flatten 70
8.1.11. intersect 70
8.1.12. project 70
8.1.13. remove-keys 71
8.1.14. values 72
8.1.15. encode-for-roundtrip 72
8.1.16. decode-from-roundtrip 72
8.2. Functions taken from XQuery 72

9. Equality and identity **77**

10. Notes **79**
10.1. Sequences vs. Arrays 79
10.2. Null vs. empty sequence 81

11. Open Issues **85**

A. Revision History **87**

Index **89**

Introduction

1.1. Why JSONiq?

JSONiq is a query and processing language specifically designed for the popular JSON data model. The main ideas behind JSONiq are based on lessons learned in more than 30 years of relational query systems and more than 15 years of experience with designing and implementing query languages for semi-structured data like XML and RDF.

The main source of inspiration behind JSONiq is XQuery, which has been proven so far a successful and productive query language for semi-structured data (in particular XML). JSONiq borrowed a large numbers of ideas from XQuery, like the structure and semantics of a FLWOR construct, the functional aspect of the language, the semantics of comparisons in the face of data heterogeneity, the declarative, snapshot-based updates. However, unlike XQuery, JSON is not concerned with the peculiarities of XML, like mixed content, ordered children, the confusion between attributes and elements, the complexities of namespaces and QNames, or the complexities of XML Schema, and so on.

The power of the XQuery's FLWOR construct and the functional aspect, combined with the simplicity of the JSON data model result in a clean, sleek and easy to understand data processing language. As a matter of fact, JSONiq is a language that can do more than queries: it can describe powerful data processing programs, from transformations, selections, joins of heterogeneous data sets, data enrichment, information extraction, information cleaning, and so on.

Technically, the main characteristics of JSONiq (and XQuery) are the following:

- It is a *set-oriented language*. While most programming languages are designed to manipulate one object at a time, JSONiq is designed to process sets (actually, sequences) of data objects.
- It is a *functional language*. A JSONiq program is an expression; the result of the program is the result of the evaluation of the expression. Expressions have fundamental role in the language: every language construct is an expression, and expressions are fully composable.
- It is a *declarative language*. A program specifies what is the result being calculated, and does not specify low level algorithms like the sort algorithm, the fact that an algorithm is executed in main memory or is external, on a single machine or parallelized on several machines, or what access patterns (aka indexes) are being used during the evaluation of the program. Such implementation decisions should be taken automatically, by an optimizer, based on the physical characteristics of the data, and of the hardware environment. Just like a traditional database would do. The language has been designed from day one with optimizability in mind.
- It is designed for *nested, heterogeneous, semi-structured data*. Data structures in JSON can be nested with arbitrary depth, do not have a specific type pattern (i.e. are heterogeneous), and may or may not have one or more schemas that describe the data. Even in the case of a schema, such a schema can be open, and/or simply partially describe the data. Unlike SQL, which is designed to query tabular, flat, homogeneous structures. JSONiq has been designed from scratch as a query for nested and heterogeneous data.

1.2. Structure of a JSONiq program.

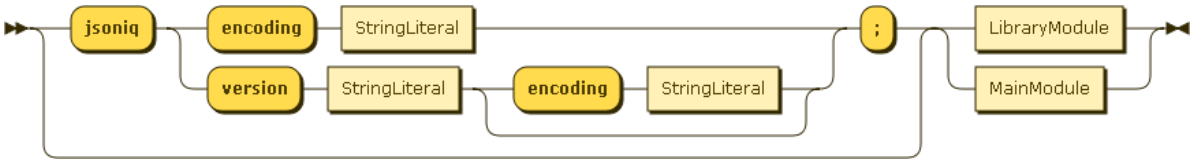

Figure 1.1. Module

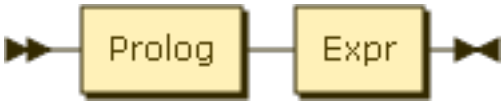

Figure 1.2. MainModule

A main JSONiq program is made of two parts: an optional prolog, and an expression, which is the main query.

The result of the main JSONiq program is the result of its main query.

In the prolog, it is possible to declare global variables and functions. Mostly, you will recognize a prolog declaration by the semi-colon it ends with. The main query does not contain semi-colons (at least in core JSONiq).

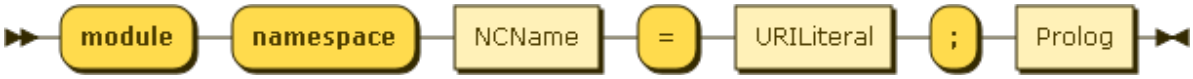

Figure 1.3. LibraryModule

Library modules do not contain any main query, just global variables and functions. They can be imported by other modules.

JSON and the JSONiq data model

JSONiq is a query language that was specifically designed for native JSON support.

2.1. JSON

As stated on json.org, JSON is a “ lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. ”

A JSON document is made of the following building blocks: objects, arrays, strings, numbers, booleans and nulls.

2.2. JSONiq values

JSONiq manipulates sequences of these building blocks, which are called items. Hence, a JSONiq value is a sequence of items.

Any JSONiq expression takes and returns sequences of items.

Comma-separated JSON-like building blocks is all you need to begin building your own sequences. You can mix and match!

Example 2.1. A sequence

```
"foo", 2, true, { "foo", "bar" }, null, [ 1, 2, 3 ]
```

Result (run with Zorba):

foo 2 true foo bar null [1, 2, 3]

Sequences are flat and cannot be nested. This makes streaming possible, which is very powerful.

Example 2.2. Sequences are flat

```
( ("foo", 2), ( (true, 4, null), 6 ) )
```

Result (run with Zorba):

foo 2 true 4 null 6

A sequence can be empty. The empty sequence can be constructed with empty parentheses.

Example 2.3. The empty sequence

```
()
```

Result (run with Zorba):

A sequence of just one item is considered the same as just this item. Whenever we say that an expression returns or takes one item, we really mean that it takes a singleton sequence of one item.

Example 2.4. A sequence of one item

```
("foo")
```

Result (run with Zorba):

foo

JSONiq classifies the items mentioned above in two categories:

- JSON items: objects and arrays.
- Atomics: strings, numbers, booleans and nulls.

2.3. Objects

An object represents a JSON object: an unordered collection of string/value pairs.

Each pair consists of an atomic of type **`string`** and of an item.

No two pairs have the same name (using Unicode character-wise comparison.)

2.4. Arrays

An array represents a JSON array: an ordered list of items.

An array can be seen as a materialized sequence, wrapped in one single item.

2.5. Atomics

An atomic is a non-structured value that is annotated with a type.

JSONiq defines the following built-in atomic types. Note that JSON numbers correspond to three different types in JSONiq.

- *string*: all JSON strings.
- *integer*: all JSON numbers that are integers (no dot, no exponent), infinite range.
- *decimal*: all JSON numbers that are decimals (no exponent), infinite range.
- *double*: IEEE double-precision 64-bit floating point numbers (corresponds to JSON numbers with an exponent).
- *boolean*: the JSON booleans true and false.
- *null*: the JSON null.

JSONiq also offers many other types of atomics. They are described in Chapter *Chapter 4, JSONiq types*.

Collections

Even though you can build your own JSON values with JSONiq by copying-and-pasting JSON documents, most of the time, your JSON data will be in a collection. Collections are sequences of objects, identified by a name which is a string.

Adding or deleting collections from the set of known collections to a query processor, and loading the data in a collection are implementation-dependent and outside of the scope of this specification.

The collection() function returns all objects associated with the provided collection name.

Example 3.1. Getting all objects from a collection

```
collection("one-object")
```

Result (run with Zorba):

{ "foo" : "bar" }

For illustrative purposes, we will assume that we have the following collections:

- *collection("one-object")*

```
{ "foo" : "bar" }
```

- *collection("captains")*

```
     { "name" : "James T. Kirk", "series" : [ "The original series" ], "century" : 23 }
     { "name" : "Jean-Luc Picard", "series" : [ "The next generation" ], "century" : 24 }
     { "name" : "Benjamin Sisko", "series" : [ "The next generation", "Deep Space 9" ],
"century" : 24 }
     { "name" : "Kathryn Janeway", "series" : [ "The next generation", "Voyager" ],
"century" : 24  }
     { "name" : "Jonathan Archer", "series" : [ "Entreprise" ], "century" : 22 }
     { "codename" : "Emergency Command Hologram", "surname" : "The Doctor", "series" :
[ "Voyager" ], "century" : 24 }
     { "name" : "Samantha Carter", "series" : [ ], "century" : 21 }
```

- *collection("films")*

```
       { "id" : "I", "name" : "The Motion Picture", "captain" : "James T. Kirk" }
       { "id" : "II", "name" : "The Wrath of Kahn", "captain" : "James T. Kirk" }
       { "id" : "III", "name" : "The Search for Spock", "captain" : "James T. Kirk" }
       { "id" : "IV", "name" : "The Voyage Home", "captain" : "James T. Kirk" }
       { "id" : "V", "name" : "The Final Frontier", "captain" : "James T. Kirk" }
       { "id" : "VI", "name" : "The Undiscovered Country", "captain" : "James T. Kirk" }
       { "id" : "VII", "name" : "Generations", "captain" : [ "James T. Kirk", "Jean-Luc
Picard" ] }
       { "id" : "VIII", "name" : "First Contact", "captain" : "Jean-Luc Picard" }
       { "id" : "IX", "name" : "Insurrection", "captain" : "Jean-Luc Picard" }
       { "id" : "X", "name" : "Nemesis", "captain" : "Jean-Luc Picard" }
       { "id" : "XI", "name" : "Star Trek", "captain" : "Spock" }
       { "id" : "XII", "name" : "Star Trek Into Darkness", "captain" : "Spock" }
```

JSONiq types

This section describes JSONiq types as well as the sequence type syntax.

JSONiq manipulates semi-structured data: in general, JSONiq allows you, but does not require you to specify types. So you have as much or as little type verification as you wish.

JSONiq is still strongly typed, so that you will be told if there is a type inconsistency or mismatch in your programs.

Whenever you do not specify the type of a variable or the type signature of a function, the most general type for any sequence of items, item*, is assumed.

Section *Section 5.7, "Expressions dealing with types"* introduces expressions which work with values of these types, as well as type operations (variable types, casts, ...).

4.1. Item types

4.1.1. Atomics

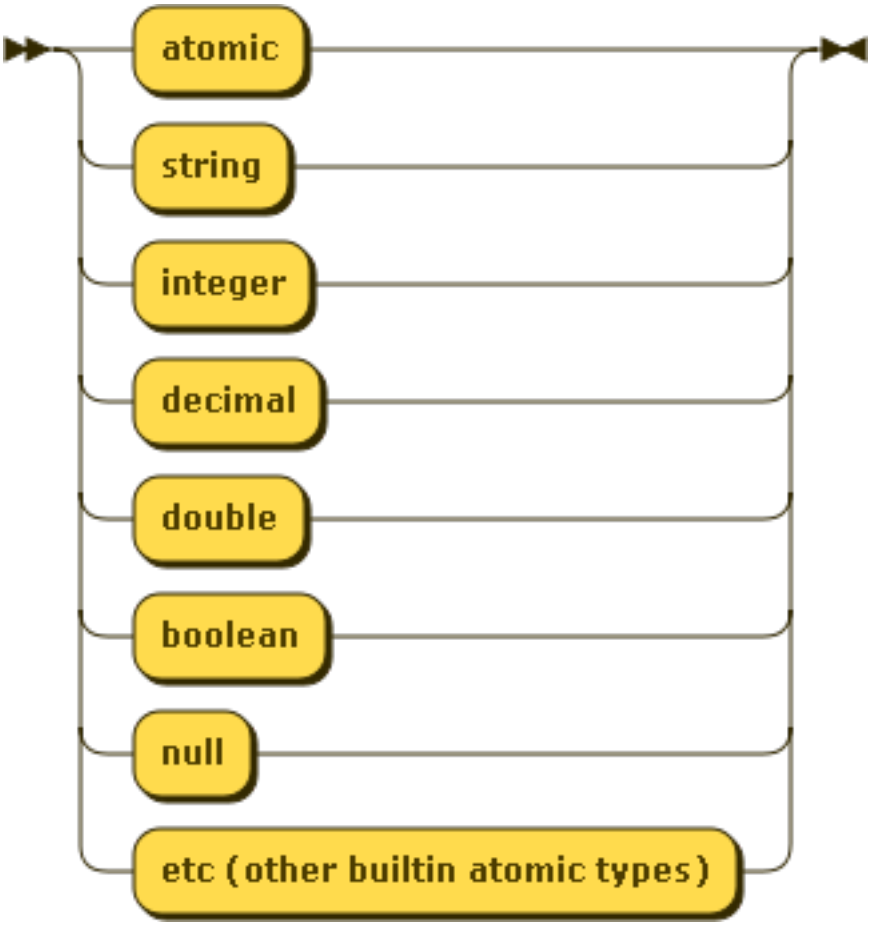

Figure 4.1. AtomicOrUnionType

Atomic types are organized in a tree hierarchy.

JSONiq defines the following build-in types that have a direct relation with JSON:

- *string*: the value space is all strings made of Unicode characters.

  All string literals build an atomic which matches string.

- *integer*: the value space is that of all mathematical integral numbers (N), with an infinite range. This is a subtype of *decimal*, so that all integers also match the item type *decimal*.

  All integer literals build an atomic which matches integer.

- *decimal*: the value space is that of all mathematical decimal numbers (D), with an infinite range.

  All decimal literals build an atomic which matches decimal.

- *double*: the value space is that of all IEEE double-precision 64-bit floating point numbers.

  All double literals build an atomic which matches double.

- *boolean*: the value space contains the booleans true and false.

  All boolean literals build an atomic which matches boolean.

- *null*: the value space is a singleton and only contains null.

  All null literals build an atomic which matches null.

- *atomic*: all atomic types.

  All literals build an atomic which matches atomic.

JSONiq also supports further atomic types, which were borrowed from *XML Schema*[1].

These datatypes are already used as a set of atomic datatypes by the other two semi-structured data formats of the Web: XML and RDF, as well as by the corresponding query languages: XQuery and SPARQL, so it is natural for a complete JSON data model to reuse them.

- Further number types: long, int, short, byte, float.
- Date or time types: date, dateTime, dateTimeStamp, gDay, gMonth, gMonthDay, gYear, gYearMonth, time.
- Duration types: duration, dayTimeDuration, yearMonthDuration.
- Binary types: base64Binary, hexBinary.
- An URI type: anyURI.

[1] http://www.w3.org/TR/xmlschema11-2/#built-in-datatypes

4.1.2. JSON items

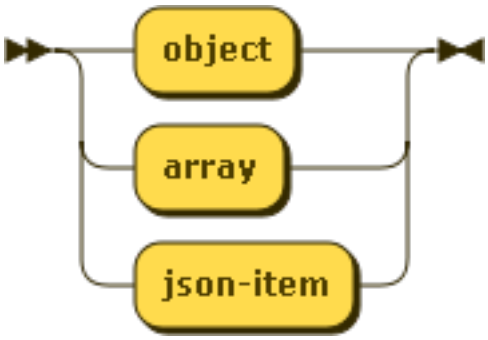

Figure 4.2. JSONItemTest

All objects match the item type *object* as well as *json-item*.

All arrays match the item type *array* as well as *json-item*.

4.1.3. The most general item type.

All items match the item type *item*.

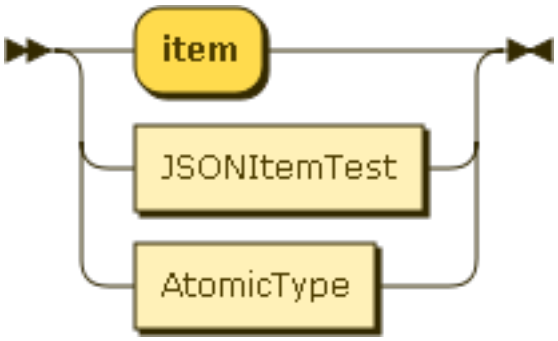

Figure 4.3. ItemType

4.2. Sequence types

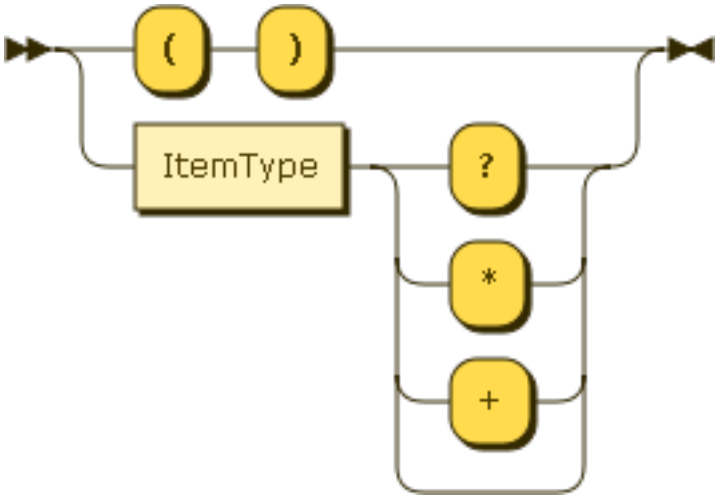

Figure 4.4. SequenceType

A sequence is an ordered list of items.

All sequences match the sequence type *item**.

A sequence type is made of an item type followed by an occurrence indicator:

- * stands for a sequence of any length (zero or more)
- + stands for a non-empty sequence (one or more)
- ? stands for an empty or a singleton sequence (zero or one)
- The absence of indicator stands for a singleton sequence (one).

Examples:

- string matches any singleton sequence containing a string.
- item+ matches any non-empty sequence.
- object? matches the empty sequence and any sequence containing one object.

Expressions

5.1. Construction of items

This will not come as a surprise since JSONiq is tailor-made: the items mentioned in the former section are constructed... exactly as they are constructed in JSON. Yes, you already know some JSONiq: any JSON document is also a valid JSONiq query which just "returns itself"!

5.1.1. Atomic literals

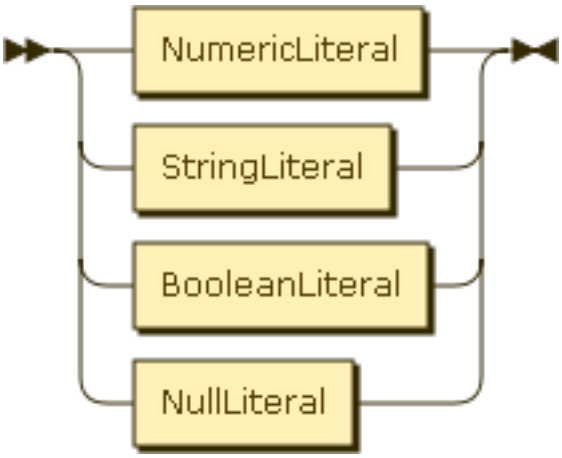

Figure 5.1. Literal

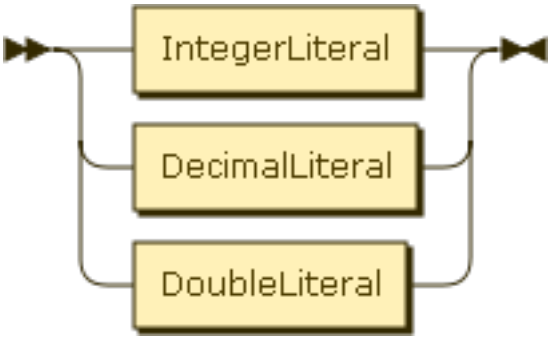

Figure 5.2. NumericLiteral

Figure 5.3. IntegerLiteral

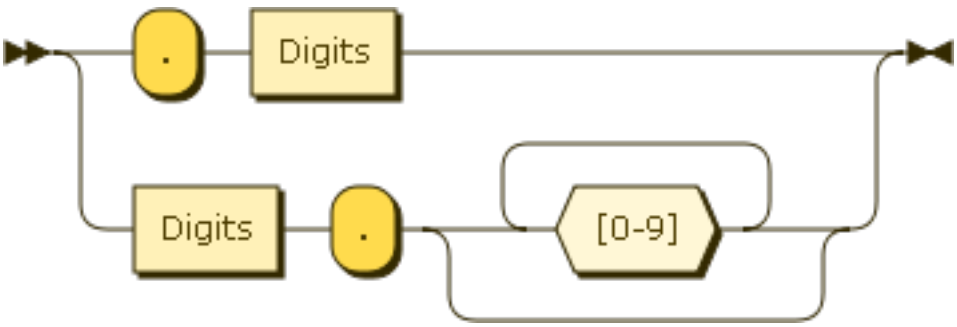

Figure 5.4. DecimalLiteral

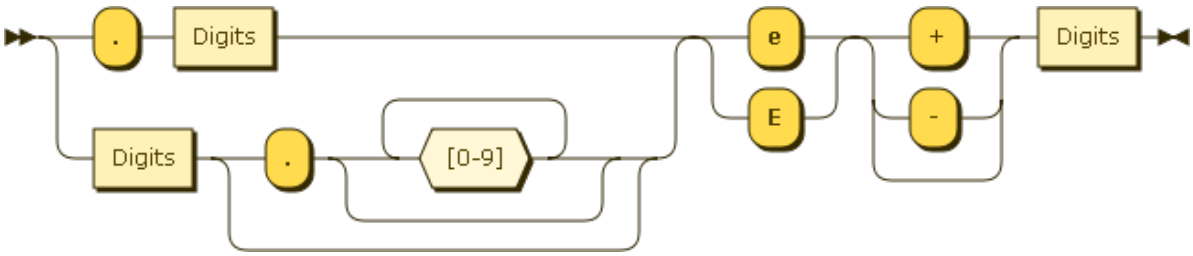

Figure 5.5. DoubleLiteral

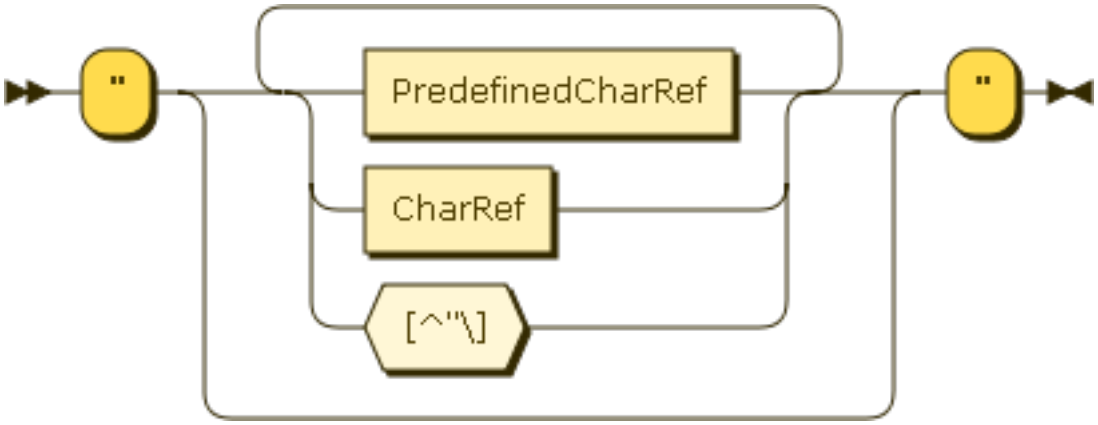

Figure 5.6. StringLiteral

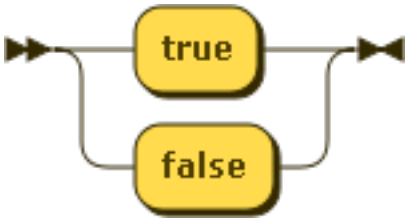

Figure 5.7. BooleanLiteral

Figure 5.8. NullLiteral

The syntax for creating strings, numbers, booleans and nulls is identical to that of JSON (it is actually a more flexible superset, for example leading 0s are allowed, and a decimal literal can begin with a dot). Note that JSONiq distinguishes between integers (no dot, no scientific notation), decimals (dot but no scientific notation) and doubles (scientific notation). As expected, an integer literal creates an atomic of type integer, and so on. No surprises. JSON's backslash escaping is supported in string literals. Also, like in JSON, double quotes are required and single quotes are forbidden.

Example 5.1. String literals

```
"foo"
```

Result (run with Zorba):

foo

Example 5.2. String literals with escaping

```
"This is a line\nand this is a new line"
```

Result (run with Zorba):

This is a line and this is a new line

Example 5.3. String literals with Unicode character escaping

```
"\u0001"
```

Result (run with Zorba):

&#x1;

Example 5.4. String literals with a nested quote

```
"This is a nested \"quote\""
```

Result (run with Zorba):

This is a nested "quote"

Example 5.5. Integer literals

```
42
```

Result (run with Zorba):

42

Example 5.6. Decimal literals

```
3.14
```

Result (run with Zorba):

3.14

Example 5.7. Double literals

```
+6.022E23
```

Result (run with Zorba):

6.022E23

Example 5.8. Boolean literals (true)

```
true
```

Result (run with Zorba):

true

Example 5.9. Boolean literals (false)

```
false
```

Result (run with Zorba):

false

Example 5.10. Null literals

```
null
```

Result (run with Zorba):

null

5.1.2. Object constructors

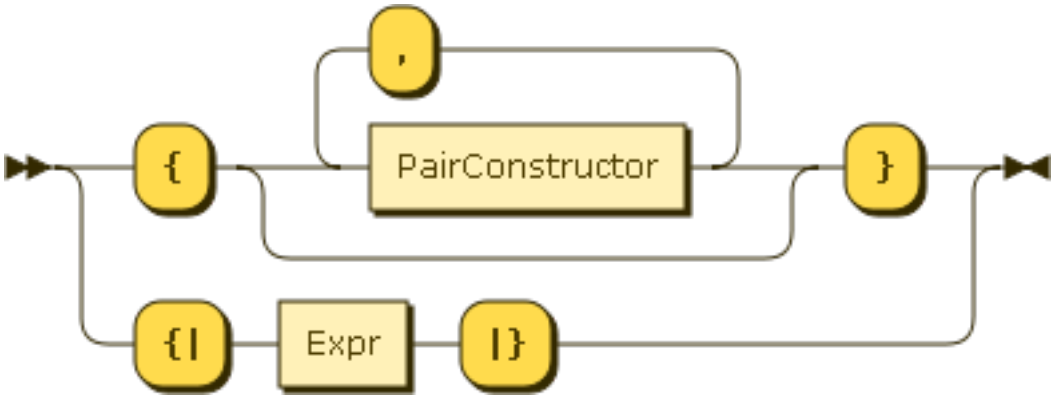

Figure 5.9. ObjectConstructor

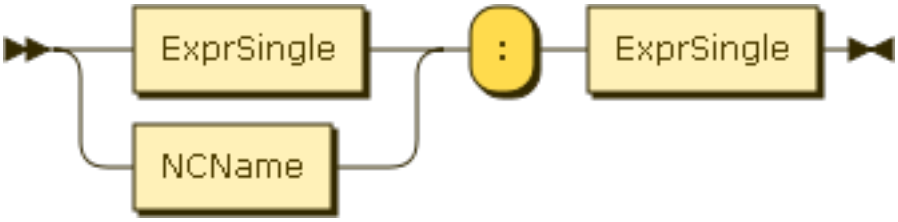

Figure 5.10. PairConstructor

The syntax for creating objects is identical to that of JSON. You can use for an object key any string literal, and for an object value any literal, object constructor or array constructor.

Example 5.11. Empty object constructors

```
{}
```

Result (run with Zorba):

{ }

Example 5.12. Object constructors 1

```
{ "foo" : "bar" }
```

Result (run with Zorba):

{ "foo" : "bar" }

Example 5.13. Object constructors 2

```
{ "foo" : [ 1, 2, 3, 4, 5, 6 ] }
```

Result (run with Zorba):

{ "foo" : [1, 2, 3, 4, 5, 6] }

Example 5.14. Object constructors 3

```
{ "foo" : true, "bar" : false }
```

Result (run with Zorba):

{ "foo" : true, "bar" : false }

Example 5.15. Nested object constructors

```
{ "this is a key" : { "value" : "a value" } }
```

Result (run with Zorba):

{ "this is a key" : { "value" : "a value" } }

Oh, and like in JavaScript, if your key is simple enough (like alphanumerics, underscores, dashes, this kind of things), you are welcome to omit the quotes. The strings for which quotes are not mandatory are called NCNames. This class of strings can be used for unquoted keys, for variable and function names, and for module aliases.

Example 5.16. Object constructors with unquoted key 1

```
{ foo : "bar" }
```

Result (run with Zorba):

{ "foo" : "bar" }

Example 5.17. Object constructors with unquoted key 2

```
{ foo : [ 1, 2, 3, 4, 5, 6 ] }
```

Result (run with Zorba):

{ "foo" : [1, 2, 3, 4, 5, 6] }

Example 5.18. Object constructors with unquoted key 3

```
{ foo : "bar", bar : "foo" }
```

Result (run with Zorba):

{ "foo" : "bar", "bar" : "foo" }

Example 5.19. Object constructors with needed quotes around the key

```
{ "but you need the quotes here" : null }
```

Result (run with Zorba):

{ "but you need the quotes here" : null }

5.1.3. Array constructors

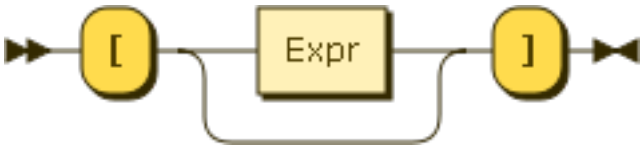

Figure 5.11. ArrayConstructor

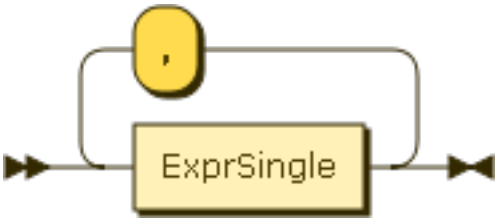

Figure 5.12. Expr

The syntax for creating arrays is identical to that of JSON: square brackets, comma separated literals, object constructors and arrays constructors.

Example 5.20. Empty array constructors

```
[]
```

Result (run with Zorba):

[ ]

Example 5.21. Array constructors

```
[ 1, 2, 3, 4, 5, 6 ]
```

Result (run with Zorba):

[1, 2, 3, 4, 5, 6]

Example 5.22. Nested array constructors

```
[ "foo", 3.14, [ "Go", "Boldly", "When", "No", "Man", "Has", "Gone", "Before" ], { "foo" :
 "bar" }, true, false, null ]
```

Result (run with Zorba):

["foo", 3.14, ["Go", "Boldly", "When", "No", "Man", "Has", "Gone", "Before"], { "foo" : "bar" }, true, false, null]

Square brackets are mandatory. Do not push it.

5.1.4. Composing constructors

Of course, JSONiq would not be very interesting if all you could do is copy and paste JSON documents.

Because JSONiq expressions are fully composable, in objects and arrays constructors, you can put way more than just atomic literals, object constructors and array constructors: you can put any JSONiq expression.

The following examples are just a few of many operators you can use: "to" for creating arithmetic sequences, "||" for concatenating strings, "+" for adding numbers, "," for appending sequences.

In an array, the operand expression will evaluated to a sequence of items, and these items will be copied and become members of the newly created array.

Example 5.23. Composable array constructors

```
[ 1 to 10 ]
```

Result (run with Zorba):

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In an object, the expression you use for the key must evaluate to an atomic - if it is not a string, it will just be cast to it.

Example 5.24. Composable object keys

```
{ "foo" || "bar" : true }
```

Result (run with Zorba):

{ "foobar" : true }

An error is raised if the key expressions is not an atomic.

Example 5.25. Non-atomic object keys

```
{ [ 1, 2 ] : true }
```

Result (run with Zorba):

An error was raised: can not atomize an array item: an array has probably been passed where an atomic value is expected (e.g., as a key, or to a function expecting an atomic item)

And do not worry about the value expression: if it is is empty, null will be used as a value, and if it contains two items or more, they will be wrapped into an array.

Example 5.26. Composable object values

```
{ "foo" : 1 + 1 }
```

Result (run with Zorba):

{ "foo" : 2 }

Example 5.27. Composable object values and automatic conversion

```
{ "foo" : (), "bar" : (1, 2) }
```

Result (run with Zorba):

{ "foo" : null, "bar" : [1, 2] }

The {| |} syntax can be used to merge several objects.

Example 5.28. Merging object constructor

```
{| { "foo" : "bar" }, { "bar" : "foo" } |}
```

Result (run with Zorba):

{ "foo" : "bar", "bar" : "foo" }

An error is raised if the operand expression does not evaluate to a sequence of objects.

Example 5.29. Merging object constructor with a type error

```
{| 1 |}
```

Result (run with Zorba):

An error was raised: xs:integer can not be treated as type object()*

5.2. Basic operations

We now introduce the most basic operations you can perform in JSONiq.

5.2.1. Construction of sequences

5.2.1.1. Comma operator

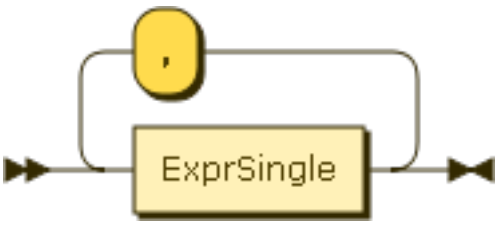

Figure 5.13. Expr

Use a comma to concatenate two sequences, or even single items. This operator has the lowest precedence of all.

Example 5.30. Comma

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Result (run with Zorba):

1 2 3 4 5 6 7 8 9 10

Example 5.31. Comma

```
{ "foo" : "bar" }, [ 1 ]
```

Result (run with Zorba):

{ "foo" : "bar" } [1]

Sequences do not nest. You need to use arrays in order to nest.

5.2.1.2. Range operator

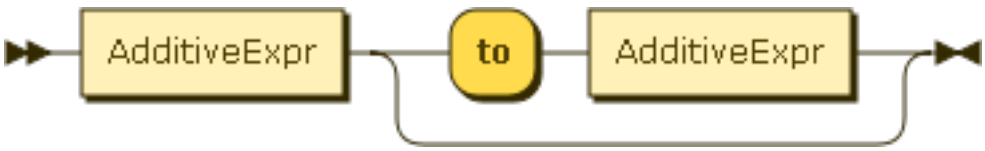

Figure 5.14. RangeExpr

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

Example 5.32. Range operator

```
1 to 10
```

Result (run with Zorba):

1 2 3 4 5 6 7 8 9 10

If one operand evaluates to the empty sequence, then the range operator returns the empty sequence.

Example 5.33. Range operator with the empty sequence

```
() to 10, 1 to ()
```

Result (run with Zorba):

Otherwise, if an operand evaluates to something else than a single integer or an empty sequence, an error is raised.

Example 5.34. Range operator with a type inconsistency

```
(1, 2) to 10
```

Result (run with Zorba):

An error was raised: sequence of more than one item can not be promoted to parameter type xs:integer? of function to()

5.2.2. Parenthesized expression

Figure 5.15. ParenthesizedExpr

Use parentheses to override the precedence of expressions.

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

Example 5.35. Empty sequence

```
()
```

Result (run with Zorba):

5.2.3. Arithmetics

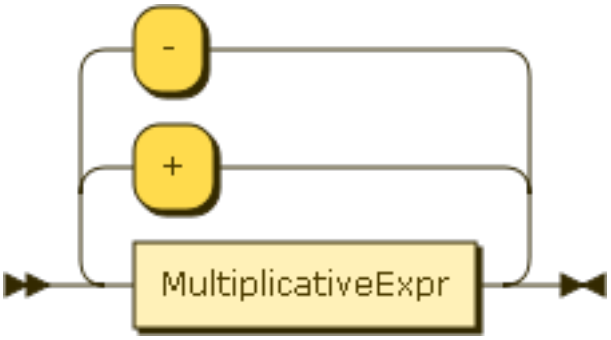

Figure 5.16. AdditiveExpr

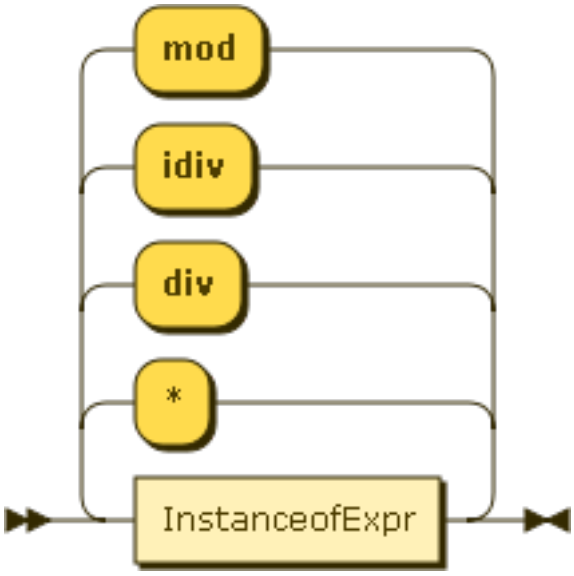

Figure 5.17. MultiplicativeExpr

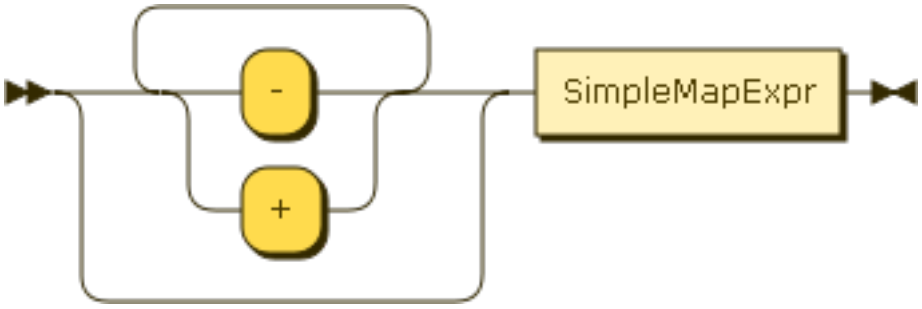

Figure 5.18. UnaryExpr

JSONiq supports the basic four operations, integer division and modulo.

Multiplicative operations have precedence over additive operations. Parentheses can override it.

Example 5.36. Basic arithmetic operations with precedence override

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

Result (run with Zorba):

8

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

Example 5.37. Using basic operations with dates.

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

Result (run with Zorba):

P29D

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

Example 5.38. Sequence of more than one number in an addition

```
(1, 2) + 3
```

Result (run with Zorba):

An error was raised: sequence of more than one item can not be promoted to parameter type xs:anyAtomicType? of function add()

If any of the operands is not a number, a date, a time or a duration, an error is raised.

Example 5.39. Null in an addition

```
1 + null
```

Result (run with Zorba):

An error was raised: arithmetic operation not defined between types "xs:integer" and "js:null"

If one of the operands evaluates to the empty sequence, then the operation results in the empty sequence.

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

Example 5.40. Basic arithmetic operations with an empty sequence

```
() + 2
```

Result (run with Zorba):

5.2.4. String concatenation

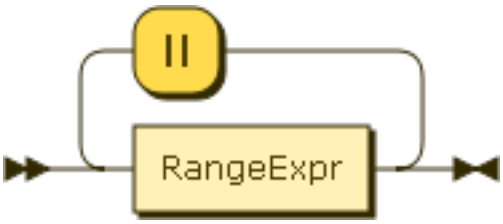

Figure 5.19. StringConcatExpr

Two strings or more can be concatenated using the concatenation operator.

Example 5.41. String concatenation

```
"Captain" || " " || "Kirk"
```

Result (run with Zorba):

Captain Kirk

An empty sequence is treated like an empty string.

Example 5.42. String concatenation with the empty sequence

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

Result (run with Zorba):

CaptainKirk

5.2.5. Comparison

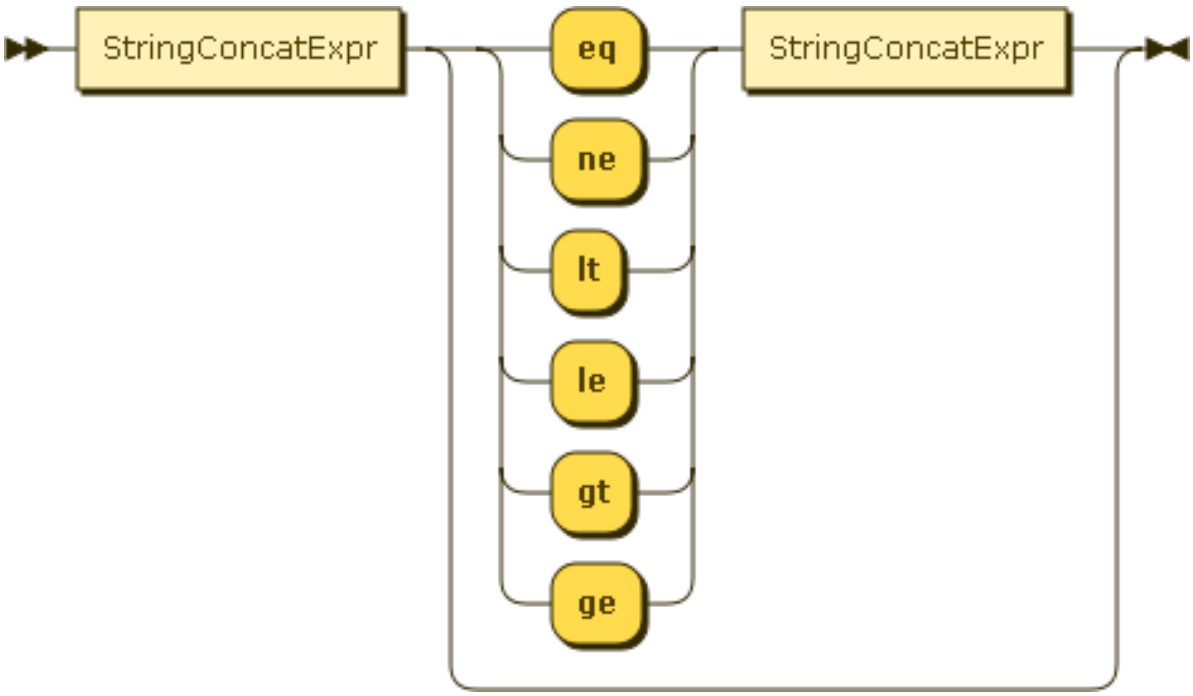

Figure 5.20. ComparisonExpr

Figure 5.21. ValueComp

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.

Example 5.43. Equality comparison

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

Result (run with Zorba):

true true

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

Example 5.44. Comparisons with a type mismatch

```
"foo" eq 1
```

Result (run with Zorba):

An error was raised: "xs:string": invalid type: can not compare for equality to type "xs:integer"

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 5.45. Equality and non-equality comparison with null

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

Result (run with Zorba):

false true true

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

Example 5.46. Ordering comparison with null

```
1 lt null
```

Result (run with Zorba):

false

Like for arithmetic operations, if an operand is the empty sequence, the empty sequence is returned as well.

Example 5.47. Comparison with the empty sequence

```
() eq 1
```

Result (run with Zorba):

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.

5.2.6. Logics

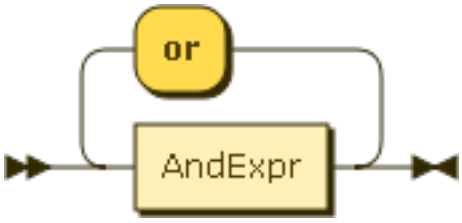

Figure 5.22. OrExpr

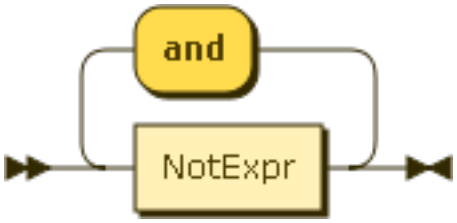

Figure 5.23. AndExpr

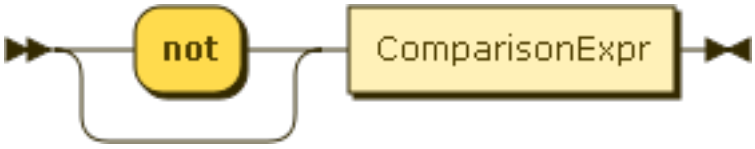

Figure 5.24. NotExpr

JSONiq logics support is based on two-valued logics: just true and false.

Non-boolean operands get automatically converted to either true or false, or an error is raised. The boolean() function performs a manual conversion.

- 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.
- An operand singleton sequence whose first item is an object or array is converted to true.
- Other operand sequences cannot be converted and an error is raised.

JSONiq supports the most famous three boolean operations: conjunction, disjunction and negation. Negation has the highest precedence, then conjunction, then disjunction. Parentheses can override.

> Example 5.48. Logics with booleans
>
> ```
> ```

```
true and ( true or not true )
```

Result (run with Zorba):

true

Example 5.49. Logics with comparing operands

```
1 + 1 eq 2 or 1 + 1 eq 3
```

Result (run with Zorba):

true

Example 5.50. Conversion of the empty sequence to false

```
boolean(())
```

Result (run with Zorba):

false

Example 5.51. Conversion of null to false

```
boolean(null)
```

Result (run with Zorba):

false

Example 5.52. Conversion of a string to true

```
boolean("foo"), boolean("")
```

Result (run with Zorba):

true false

Example 5.53. Conversion of a number to false

```
0 and true, not (not 1e42)
```

Result (run with Zorba):

false true

Example 5.54. Conversion of an object to a boolean (not implemented in Zorba at this point)

```
{ "foo" : "bar" } or false
```

Result (run with Zorba):

An error was raised: "[JSONXQType object]": invalid argument type for function fn:boolean()

If the input sequence has more than one item, and the first item is not an object or array, an error is raised.

Example 5.55. Error upon conversion of a sequence of more than one item, not beginning with a JSON item, to a boolean

```
( 1, 2, 3 ) or false
```

Result (run with Zorba):

An error was raised: invalid argument type for function fn:boolean(): effective boolean value not defined for sequence of more than one item that starts with "xs:integer"

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 5.56. Non-determinism in presence of errors.

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

Result (run with Zorba):

true

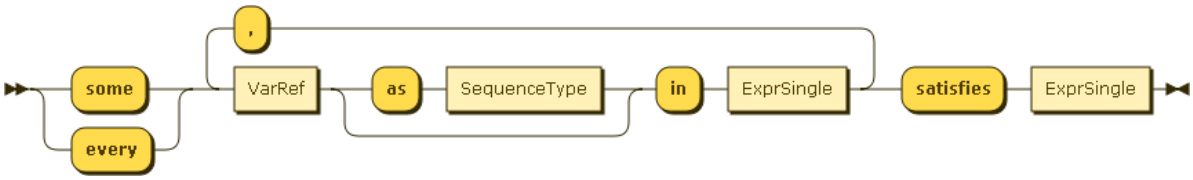

Figure 5.25. QuantifiedExpr

It is possible to perform a conjunction or a disjunction on a predicate for each item in a sequence.

Example 5.57. Universal quantifier

```
every $i in 1 to 10 satisfies $i gt 0
```

Result (run with Zorba):

true

Example 5.58. Existential quantifier on several variables

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

Result (run with Zorba):

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 5.59. 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 (run with Zorba):

true

5.3. Function Calls

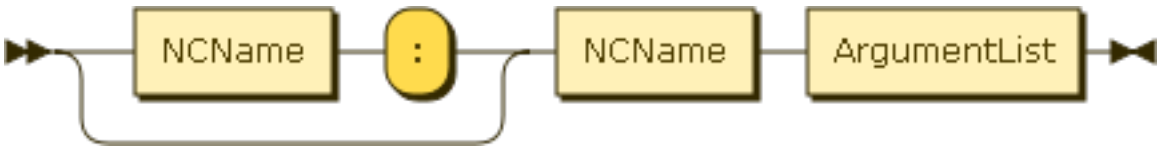

Figure 5.26. FunctionCall

The syntax for function calls is similar to many other languages. 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 *Chapter 6, Prologs* 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 *Chapter 7, Modules* describes how to define your own modules.

Example 5.60. A builtin function call.

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

Result (run with Zorba):

foo bar

Example 5.61. A builtin function call.

```
concat("foo", "bar")
```

Result (run with Zorba):

foobar

An error is raised if the actual types do not match the expected types.

Example 5.62. A type error in a function call.

```
sum({ "foo" : "bar" })
```

Result (run with Zorba):

An error was raised: can not atomize an object item: an object has probably been passed where an atomic value is expected (e.g., as a key, or to a function expecting an atomic item)

5.4. Selectors

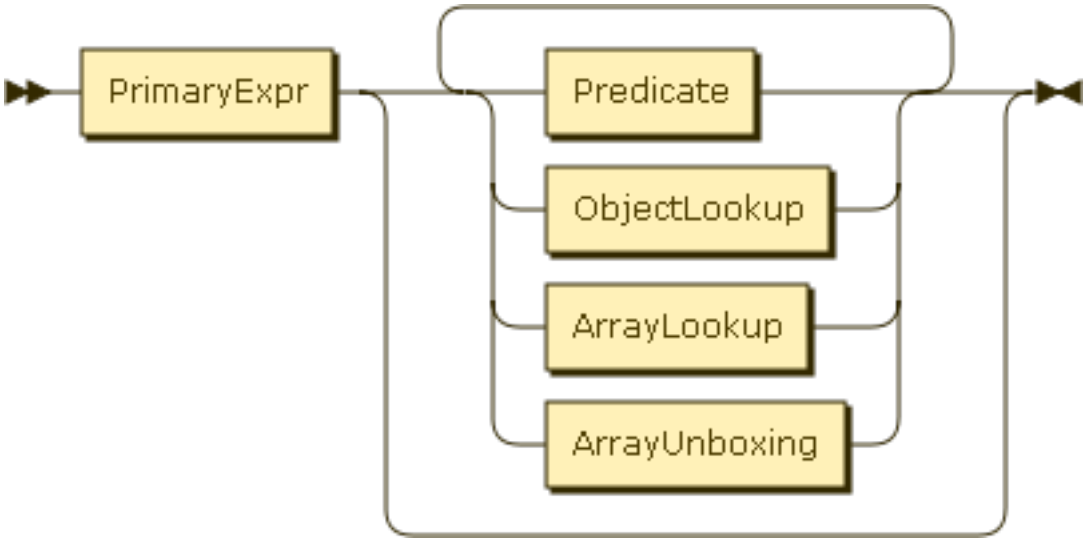

Figure 5.27. PostfixExpr

Like in JavaScript, it is possible to navigate through data.

JSONiq supports filtering items from a sequence, looking up the value associated with a given key in an object, looking up the item at a given position in an array, and looking up all items in an array.

5.4.1. Object field selector

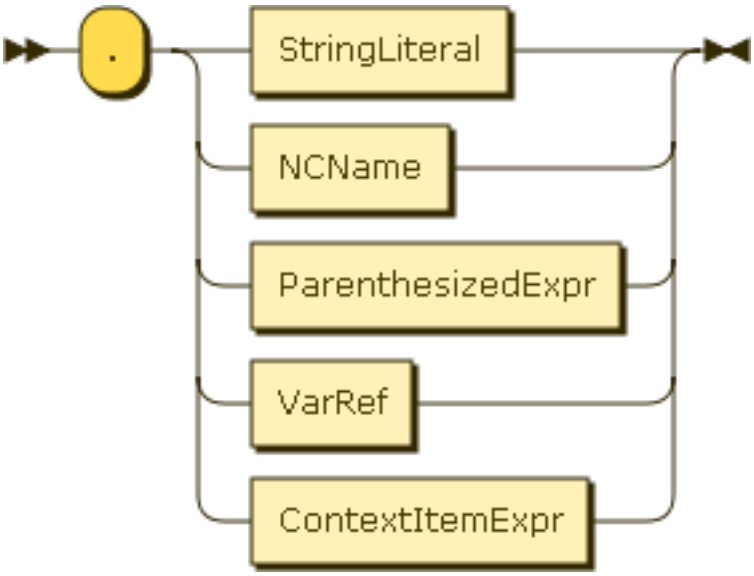

Figure 5.28. ObjectLookup

The simplest way to navigate in an object is similar to JavaScript, using a dot. This will work as soon as you do not push it too much: alphanumerical characters, dashes, underscores - just like unquoted keys in object constructors, any NCName is allowed.

Example 5.63. Object lookup

```
{ "foo" : "bar" }.foo
```

Result (run with Zorba):

bar

Since JSONiq expressions are composable, you can also use any expression for the left-hand side. You might need parentheses depending on the precedence.

Example 5.64. Lookup on a single-object collection.

```
collection("one-object").foo
```

Result (run with Zorba):

bar

The dot operator does an implicit mapping on the left-hand-side, i.e., it applies the lookup in turn on each item. Lookup on an object returns the value associated with the supplied key, or the empty sequence if there is none. Lookup on any item which is not an object (arrays and atomics) results in the empty sequence.

Example 5.65. Object lookup with an iteration on several objects

```
({ "foo" : "bar" }, { "foo" : "bar2" }, { "bar" : "foo" }).foo
```

Result (run with Zorba):

bar bar2

Example 5.66. Object lookup with an iteration on a collection

```
collection("captains").name
```

Result (run with Zorba):

James T. Kirk Jean-Luc Picard Benjamin Sisko Kathryn Janeway Jonathan Archer Samantha Carter

Example 5.67. Object lookup on a mixed sequence

```
({ "foo" : "bar1" }, [ "foo", "bar" ], { "foo" : "bar2" }, "foo").foo
```

Result (run with Zorba):

bar1 bar2

Of course, unquoted keys will not work for strings that are not NCNames, e.g., if the field contains a dot or begins with a digit. Then you will need quotes.

Example 5.68. Quotes for object lookup

```
{ "foo bar" : "bar" }."foo bar"
```

Result (run with Zorba):

bar

If you use an expression on the right side of the dot, it must always have parentheses. The result of the right-hand-side expression is cast to a string. An error is raised if the cast fails.

Example 5.69. Object lookup with a nested expression

```
{ "foobar" : "bar" }.("foo" || "bar")
```

Result (run with Zorba):

bar

Example 5.70. Object lookup with a nested expression

```
{ "foobar" : "bar" }.("foo", "bar")
```

Result (run with Zorba):

An error was raised: sequence of more than one item can not be treated as type xs:string

Example 5.71. Object lookup with a nested expression

```
{ "1" : "bar" }.(1)
```

Result (run with Zorba):

bar

Variables, or a context item reference, do not need parentheses. Variables are introduced later, but here is a sneak peek:

Example 5.72. Object lookup with a variable

```
let $field := "foo" || "bar"
return { "foobar" : "bar" }.$field
```

Result (run with Zorba):

bar

5.4.2. Array member selector

Figure 5.29. ArrayLookup

Array lookup uses double square brackets.

Example 5.73. Array lookup

```
[ "foo", "bar" ] [[2]]
```

Result (run with Zorba):

bar

Since JSONiq expressions are composable, you can also use any expression for the left-hand side. You might need parentheses depending on the precedence.

Example 5.74. Array lookup after an object lookup

```
{ field : [ "one",  { "foo" : "bar" } ] }.field[[2]].foo
```

Result (run with Zorba):

bar

The array lookup operator does an implicit mapping on the left-hand-side, i.e., it applies the lookup in turn on each item. Lookup on an array returns the item at that position in the array, or the empty sequence if there is none (position larger than size or smaller than 1). Lookup on any item which is not an array (objects and atomics) results in the empty sequence.

Example 5.75. Array lookup with an iteration on several arrays

```
([ 1, 2, 3 ], [ 4, 5, 6 ])[[2]]
```

Result (run with Zorba):

2 5

Example 5.76. Array lookup with an iteration on a collection

```
collection("captains").series[[1]]
```

Result (run with Zorba):

The original series The next generation The next generation The next generation Entreprise Voyager

Example 5.77. Array lookup on a mixed sequence

```
([ 1, 2, 3 ], [ 4, 5, 6 ], { "foo" : "bar" }, true)[[3]]
```

Result (run with Zorba):

3 6

The expression inside the double-square brackets may be any expression. The result of evaluating this expression is cast to an integer. An error is raised if the cast fails.

Example 5.78. Array lookup with a right-hand-side expression

```
[ "foo", "bar" ] [[ 1 + 1 ]]
```

Result (run with Zorba):

bar

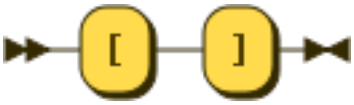

Figure 5.30. ArrayUnboxing

You can also extract all items from an array (i.e., as a sequence) with the [] syntax. The [] operator also implicitly iterates on the left-hand-side, returning the empty sequence for non-arrays.

Example 5.79. Extracting all items from an array

```
[ "foo", "bar" ][]
```

Result (run with Zorba):

foo bar

Example 5.80. Extracting all items from arrays in a mixed sequence

```
([ "foo", "bar" ], { "foo" : "bar" }, true, [ 1, 2, 3 ] )[]
```

Result (run with Zorba):

foo bar 1 2 3

5.4.3. Sequence predicates

Figure 5.31. Predicate

A predicate allows filtering a sequence, keeping only items that fulfill it.

The predicate is evaluated once for each item in the left-hand-side sequence, with the context item set to that item. The predicate expression can use $$ to access this context item.

Figure 5.32. ContextItemExpr

If the predicate evaluates to an integer, it is matched against the item position in the left-hand side sequence automatically

Example 5.81. Predicate expression

```
(1 to 10)[2]
```

Result (run with Zorba):

2

Otherwise, the result of the predicate is converted to a boolean.

All items for which the converted predicate result evaluates to true are then output.

Example 5.82. Predicate expression

```
(1 to 10)[$$ mod 2 eq 0]
```

Result (run with Zorba):

2 4 6 8 10

5.5. Control flow expressions

JSONiq supports control flow expressions such as if-then-else, switch and typeswitch.

5.5.1. Conditional expressions

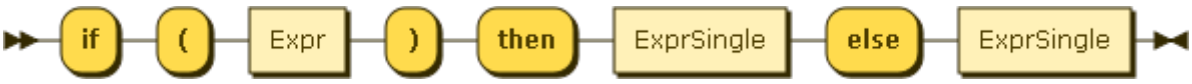

Figure 5.33. IfExpr

A conditional expressions allows you to pick one or another value depending on a boolean value.

Example 5.83. A conditional expression

```
if (1 + 1 eq 2) then { "foo" : "yes" } else { "foo" : "false" }
```

Result (run with Zorba):

{ "foo" : "yes" }

The behavior of the expression inside the if is similar to that of logical operations (two-valued logics), meaning that non-boolean values get converted to a boolean.

Example 5.84. A conditional expression

```
if (null) then { "foo" : "yes" } else { "foo" : "no" }
```

Result (run with Zorba):

{ "foo" : "no" }

Example 5.85. A conditional expression

```
if (1) then { "foo" : "yes" } else { "foo" : "no" }
```

Result (run with Zorba):

{ "foo" : "yes" }

Example 5.86. A conditional expression

```
if (0) then { "foo" : "yes" } else { "foo" : "no" }
```

Result (run with Zorba):

{ "foo" : "no" }

Example 5.87. A conditional expression

```
if ("foo") then { "foo" : "yes" } else { "foo" : "no" }
```

Result (run with Zorba):

{ "foo" : "yes" }

Example 5.88. A conditional expression

```
if ("") then { "foo" : "yes" } else { "foo" : "no" }
```

Result (run with Zorba):

{ "foo" : "no" }

Example 5.89. A conditional expression

```
if (()) then { "foo" : "yes" } else { "foo" : "no" }
```

Result (run with Zorba):

{ "foo" : "no" }

Example 5.90. A conditional expression

```
if (({ "foo" : "bar" }, [ 1, 2, 3, 4])) then { "foo" : "yes" } else { "foo" : "no" }
```

Result (run with Zorba):

An error was raised: "[JSONXQType object]": invalid argument type for function fn:boolean()

Note that the else clause is mandatory (but can be the empty sequence)

Example 5.91. A conditional expression

```
if (1+1 eq 2) then { "foo" : "yes" } else ()
```

Result (run with Zorba):

{ "foo" : "yes" }

5.5.2. Switch expressions

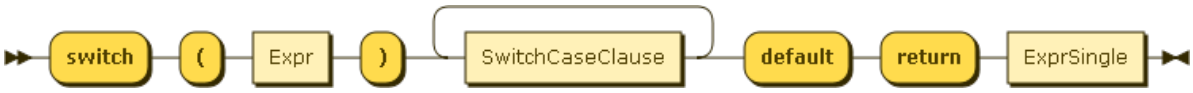

Figure 5.34. SwitchExpr

A switch expression evaluates the expression inside the switch. If it is an atomic, it compares it in turn to the provided atomic values (with the semantics of the eq operator) and returns the value associated with the first matching case clause.

Example 5.92. A switch expression

```
switch ("foo")
case "bar" return "foo"
case "foo" return "bar"
default return "none"
```

Result (run with Zorba):

bar

If it is not an atomic, an error is raised.

Example 5.93. A switch expression

```
switch ({ "foo" : "bar" })
case "bar" return "foo"
case "foo" return "bar"
default return "none"
```

Result (run with Zorba):

An error was raised: can not atomize an object item: an object has probably been passed where an atomic value is expected (e.g., as a key, or to a function expecting an atomic item)

If no value matches, the default is used.

Example 5.94. A switch expression

```
switch ("no-match")
case "bar" return "foo"
case "foo" return "bar"
default return "none"
```

Result (run with Zorba):

none

The case clauses support composability of expressions as well.

Example 5.95. A switch expression

```
switch (2)
case 1 + 1 return "foo"
case 2 + 2 return "bar"
default return "none"
```

Result (run with Zorba):

foo

Example 5.96. A switch expression

```
switch (true)
case 1 + 1 eq 2 return "1 + 1 is 2"
case 2 + 2 eq 5 return "2 + 2 is 5"
default return "none of the above is true"
```

Result (run with Zorba):

1 + 1 is 2

5.5.3. Try-catch expressions

Figure 5.35. TryCatchExpr

A try catch expression evaluates the expression inside the try block and returns its resulting value.

However, if an error is raised dynamically, the catch clause is evaluated and its result value returned.

Example 5.97. A try catch expression

```
try { 1 div 0 } catch * { "division by zero!" }
```

Result (run with Zorba):

division by zero!

Only errors raised within the lexical scope of the try block are caught.

Example 5.98. A try catch expression

```
let $x := 1 div 0
return try { $x }
       catch * { "division by zero!" }
```

Result (run with Zorba):

An error was raised: division by zero

Errors that are detected statically within the try block are still reported statically.

Example 5.99. A try catch expression

```
try { x } catch * { "syntax error" }
```

Result (run with Zorba):

An error was raised: invalid expression: syntax error, a path expression cannot begin with an axis step

5.6. FLWOR expressions

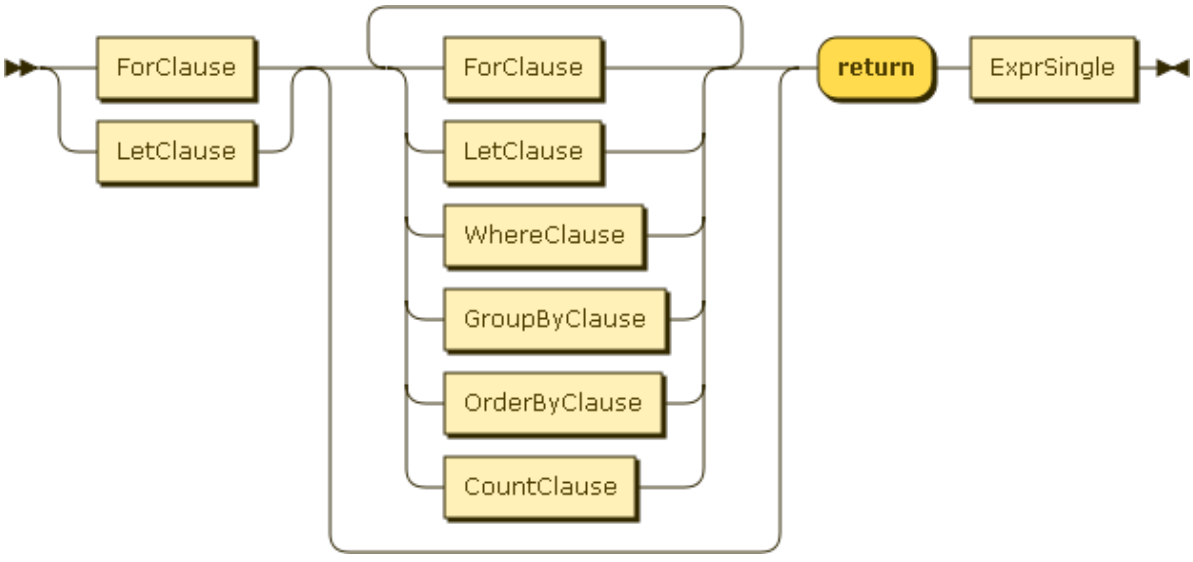

Figure 5.36. FLWORExpr

FLWOR expressions are probably the most powerful JSONiq construct and correspond to SQL's SELECT-FROM-WHERE statements, but they are more general and more flexible. In particular, clauses can almost appear in any order (apart that it must begin with a for or let clause, and end with a return clause).

Here is a bit of theory on how it works.

A clause binds values to some variables according to its own semantics, possibly several times. Each time, a tuple of variable bindings (mapping variable names to sequences) is passed on to the next clause.

This goes all the way down, until the return clause. The return clause is eventually evaluated for each tuple of variable bindings, resulting in a sequence of items for each tuple.

These sequences of items are concatenated, in the order of the incoming tuples, and the obtained sequence is returned by the FLWOR expression.

We are now giving practical examples with a hint on how it maps to SQL.

5.6.1. For clauses

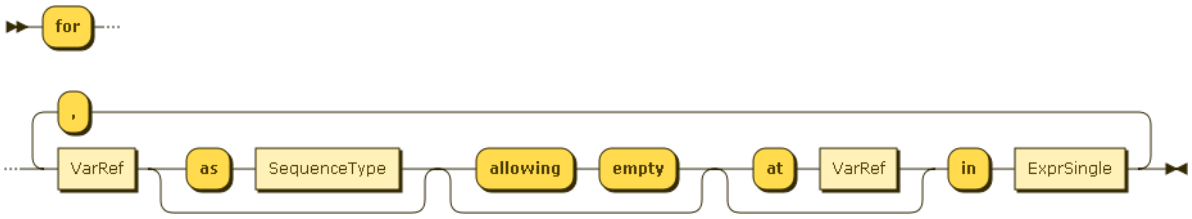

Figure 5.37. ForClause

For clauses allow iteration on a sequence.

For each incoming tuple, the expression in the for clause is evaluated to a sequence. Each item in this sequence is in turn bound to the for variable. A tuple is hence produced for each incoming tuple, and for each item in the sequence produced by the for clause for this tuple.

The order in which items are bound by the for clause can be relaxed with unordered expressions, as described later in this section.

The following query, using a for and a return clause, is the counterpart of SQL's "SELECT name FROM captains". $x is bound in turn to each item in the captains collection.

Example 5.100. A for clause.

```
for $x in collection("captains")
return $x.name
```

Result (run with Zorba):

James T. Kirk Jean-Luc Picard Benjamin Sisko Kathryn Janeway Jonathan Archer Samantha Carter

For clause expressions are composable, there can be several of them.

Example 5.101. Two for clauses.

```
for $x in ( 1, 2, 3 )
for $y in ( 1, 2, 3 )
return 10 * $x + $y
```

Result (run with Zorba):

11 12 13 21 22 23 31 32 33

Example 5.102. A for clause.

```
for $x in ( 1, 2, 3 ), $y in ( 1, 2, 3 )
return 10 * $x + $y
```

Result (run with Zorba):

11 12 13 21 22 23 31 32 33

A for variable is visible to subsequence bindings.

Example 5.103. A for clause.

```
for $x in ( [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ), $y in $x[]
return $y
```

Result (run with Zorba):

1 2 3 4 5 6 7 8 9

Example 5.104. A for clause.

```
for $x in collection("captains"), $y in $x.series[]
return { "captain" : $x.name, "series" : $y }
```

Result (run with Zorba):

{ "captain" : "James T. Kirk", "series" : "The original series" } { "captain" : "Jean-Luc Picard", "series" : "The next generation" } { "captain" : "Benjamin Sisko", "series" : "The next generation" } { "captain" : "Benjamin Sisko", "series" : "Deep Space 9" } { "captain" : "Kathryn Janeway", "series" : "The next generation" } { "captain" : "Kathryn Janeway", "series" : "Voyager" } { "captain" : "Jonathan Archer", "series" : "Entreprise" } { "captain" : null, "series" : "Voyager" }

It is also possible to bind the position of the current item in the sequence to a variable.

Example 5.105. A for clause.

```
for $x at $position in collection("captains")
return { "captain" : $x.name, "id" : $position }
```

Result (run with Zorba):

{ "captain" : "James T. Kirk", "id" : 1 } { "captain" : "Jean-Luc Picard", "id" : 2 } { "captain" : "Benjamin Sisko", "id" : 3 } { "captain" : "Kathryn Janeway", "id" : 4 } { "captain" : "Jonathan Archer", "id" : 5 } { "captain" : null, "id" : 6 } { "captain" : "Samantha Carter", "id" : 7 }

JSONiq supports joins. For example, the counterpart of "SELECT c.name AS captain, m.name AS movie FROM captains c JOIN movies m ON c.name = m.name" is:

Example 5.106. A join

```
for $captain in collection("captains"), $movie in collection("movies")[ try { $$.captain
 eq $captain.name } catch * { false } ]
return { "captain" : $captain.name, "movie" : $movie.name }
```

Result (run with Zorba):

{ "captain" : "James T. Kirk", "movie" : "The Motion Picture" } { "captain" : "James T. Kirk", "movie" : "The Wrath of Kahn" } { "captain" : "James T. Kirk", "movie" : "The Search for Spock" } { "captain" : "James T. Kirk", "movie" : "The Voyage Home" } { "captain" : "James T. Kirk", "movie" : "The Final Frontier" } { "captain" : "James T. Kirk", "movie" : "The Undiscovered Country" } { "captain" : "Jean-Luc Picard", "movie" : "First Contact" } { "captain" : "Jean-Luc Picard", "movie" : "Insurrection" } { "captain" : "Jean-Luc Picard", "movie" : "Nemesis" }

Note how JSONiq handles semi-structured data in a flexible way.

Outer joins are also possible with "allowing empty", i.e., output will also be produced if there is no matching movie for a captain. The following query is the counterpart of "SELECT c.name AS captain, m.name AS movie FROM captains c LEFT JOIN movies m ON c.name = m.captain".

Example 5.107. A join

```
for $captain in collection("captains"), $movie allowing empty in collection("movies")[ try
 { $$.captain eq $captain.name } catch * { false } ]
return { "captain" : $captain.name, "movie" : $movie.name }
```

Result (run with Zorba):

{ "captain" : "James T. Kirk", "movie" : "The Motion Picture" } { "captain" : "James T. Kirk", "movie" : "The Wrath of Kahn" } { "captain" : "James T. Kirk", "movie" : "The Search for Spock" } { "captain" : "James T. Kirk", "movie" : "The Voyage Home" } { "captain" : "James T. Kirk", "movie" : "The Final Frontier" } { "captain" : "James T. Kirk", "movie" : "The Undiscovered Country" } { "captain" : "Jean-Luc Picard", "movie" : "First Contact" } { "captain" : "Jean-Luc Picard", "movie" : "Insurrection" } { "captain" : "Jean-Luc Picard", "movie" : "Nemesis" } { "captain" : "Benjamin Sisko", "movie" : null } { "captain" : "Kathryn Janeway", "movie" : null } { "captain" : "Jonathan Archer", "movie" : null } { "captain" : null, "movie" : null } { "captain" : "Samantha Carter", "movie" : null }

5.6.2. Where clauses

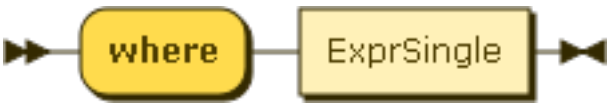

Figure 5.38. WhereClause

Where clauses are used for filtering (selection operator in the relational algebra).

For each incoming tuple, the expression in the where clause is evaluated to a boolean (possibly converting an atomic to a boolean). if this boolean is true, the tuple is forwarded to the next clause, otherwise it is dropped.

The following query corresponds to "SELECT series FROM captains WHERE name = 'Kathryn Janeway'".

Example 5.108. A where clause.

```
for $x in collection("captains")
where $x.name eq "Kathryn Janeway"
return $x.series
```

Result (run with Zorba):

["The next generation", "Voyager"]

5.6.3. Order clauses

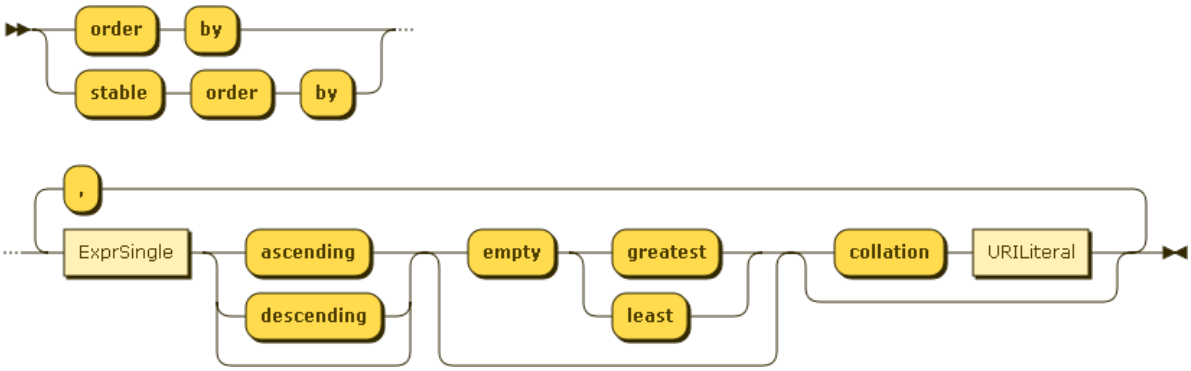

Figure 5.39. OrderByClause

Order clauses are for reordering tuples.

For each incoming tuple, the expression in the where clause is evaluated to an atomic. The tuples are then sorted based on the atomics they are associated with, and then forwarded to the next clause.

Like for ordering comparisons, null values are always considered the smallest.

The following query is the counterpart of SQL's "SELECT * FROM captains ORDER BY name".

Example 5.109. An order by clause.

```
for $x in collection("captains")
order by $x.name
return $x
```

Result (run with Zorba):

{ "name" : "Benjamin Sisko", "series" : ["The next generation", "Deep Space 9"], "century" : 24 } { "name" : "James T. Kirk", "series" : ["The original series"], "century" : 23 } { "name" : "Jean-Luc Picard", "series" : ["The next generation"], "century" : 24 } { "name" : "Jonathan Archer", "series" :

["Entreprise"], "century" : 22 } { "name" : "Kathryn Janeway", "series" : ["The next generation", "Voyager"], "century" : 24 } { "name" : "Samantha Carter", "series" : [ ], "century" : 21 } { "codename" : "Emergency Command Hologram", "surname" : "The Doctor", "series" : ["Voyager"], "century" : 24 }

Multiple sorting criteria can be given - they are treated like a lexicographic order (most important criterium first).

Example 5.110. An order by clause.

```
for $x in collection("captains")
order by size($x.series), $x.name
return $x
```

Result (run with Zorba):

{ "name" : "Samantha Carter", "series" : [ ], "century" : 21 } { "name" : "James T. Kirk", "series" : ["The original series"], "century" : 23 } { "name" : "Jean-Luc Picard", "series" : ["The next generation"], "century" : 24 } { "name" : "Jonathan Archer", "series" : ["Entreprise"], "century" : 22 } { "codename" : "Emergency Command Hologram", "surname" : "The Doctor", "series" : ["Voyager"], "century" : 24 } { "name" : "Benjamin Sisko", "series" : ["The next generation", "Deep Space 9"], "century" : 24 } { "name" : "Kathryn Janeway", "series" : ["The next generation", "Voyager"], "century" : 24 }

It can be specified whether the order is ascending or descending. Empty sequences are allowed and it can be chosen whether to put them first or last.

Example 5.111. An order by clause.

```
for $x in collection("captains")
order by $x.name descending empty greatest
return $x
```

Result (run with Zorba):

{ "codename" : "Emergency Command Hologram", "surname" : "The Doctor", "series" : ["Voyager"], "century" : 24 } { "name" : "Samantha Carter", "series" : [ ], "century" : 21 } { "name" : "Kathryn Janeway", "series" : ["The next generation", "Voyager"], "century" : 24 } { "name" : "Jonathan Archer", "series" : ["Entreprise"], "century" : 22 } { "name" : "Jean-Luc Picard", "series" : ["The next generation"], "century" : 24 } { "name" : "James T. Kirk", "series" : ["The original series"], "century" : 23 } { "name" : "Benjamin Sisko", "series" : ["The next generation", "Deep Space 9"], "century" : 24 }

An error is raised if the expression does not evaluate to an atomic or the empty sequence.

Example 5.112. An order by clause.

```
for $x in collection("captains")
order by $x
return $x.name
```

Result (run with Zorba):

An error was raised: can not atomize an object item: an object has probably been passed where an atomic value is expected (e.g., as a key, or to a function expecting an atomic item)

Collations can be used to give a specific way of how strings are to be ordered. A collation is identified by a URI.

Example 5.113. Use of a collation in an order by clause.

```
for $x in collection("captains")
order by $x.name collation "http://www.w3.org/2005/xpath-functions/collation/codepoint"
return $x.name
```

Result (run with Zorba):

Benjamin Sisko James T. Kirk Jean-Luc Picard Jonathan Archer Kathryn Janeway Samantha Carter

5.6.4. Group clauses

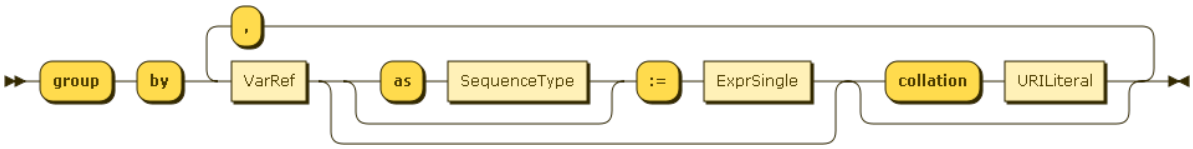

Figure 5.40. GroupByClause

Grouping is also supported, like in SQL.

For each incoming tuple, the expression in the group clause is evaluated to an atomic (a grouping key). The incoming tuples are then grouped according to the key they are associated with.

For each group, a tuple is output, with a binding from the grouping variable to the key of the group.

Example 5.114. An order by clause.

```
for $x in collection("captains")
group by $century := $x.century
return { "century" : $century  }
```

Result (run with Zorba):

{ "century" : 21 } { "century" : 22 } { "century" : 23 } { "century" : 24 }

As for the other (non-grouping) variables, their values within one group are all concatenated, keeping the same name. Aggregations can be done on these variables.

The following query is equivalent to "SELECT century, COUNT(*) FROM captains GROUP BY century".

Example 5.115. An order by clause.

```
for $x in collection("captains")
group by $century := $x.century
return { "century" : $century, "count" : count($x) }
```

Result (run with Zorba):

{ "century" : 21, "count" : 1 } { "century" : 22, "count" : 1 } { "century" : 23, "count" : 1 } { "century" : 24, "count" : 4 }

JSONiq's group by is more flexible than SQL and is fully composable.

Example 5.116. An order by clause.

```
for $x in collection("captains")
group by $century := $x.century
return { "century" : $century, "captains" : [ $x.name ] }
```

Result (run with Zorba):

{ "century" : 21, "captains" : ["Samantha Carter"] } { "century" : 22, "captains" : ["Jonathan Archer"] } { "century" : 23, "captains" : ["James T. Kirk"] } { "century" : 24, "captains" : ["Jean-Luc Picard", "Benjamin Sisko", "Kathryn Janeway"] }

Unlike SQL, JSONiq does not need a having clause, because a where clause works perfectly after grouping as well.

The following query is the counterpart of "SELECT century, COUNT(*) FROM captains GROUP BY century HAVING COUNT(*) > 1"

Example 5.117. An order by clause.

```
for $x in collection("captains")
group by $century := $x.century
where count($x) gt 1
return { "century" : $century, "count" : count($x) }
```

Result (run with Zorba):

{ "century" : 24, "count" : 4 }

5.6.5. Let clauses

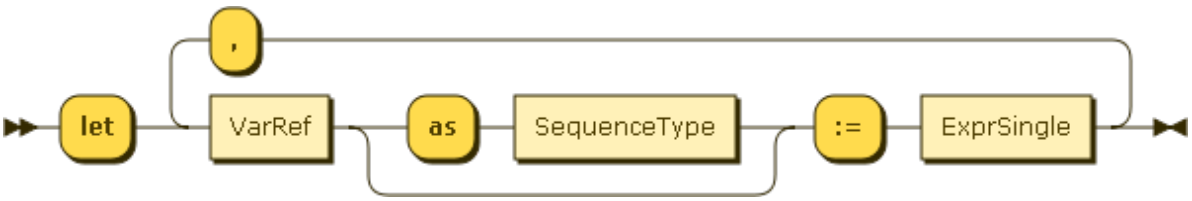

Figure 5.41. LetClause

Let bindings can be used to define aliases for any sequence, for convenience.

For each incoming tuple, the expression in the let clause is evaluated to a sequence. A binding is added from this sequence to the let variable in each tuple. A tuple is hence produced for each incoming tuple.

Example 5.118. An order by clause.

```
for $x in collection("captains")
let $century := $x.century
group by $century
let $number := count($x)
where $number gt 1
return { "century" : $century, "count" : $number }
```

Result (run with Zorba):

{ "century" : 24, "count" : 4 }

Note that it is perfectly fine to reuse a variable name and hide a variable binding.

Example 5.119. An order by clause.

```
for $x in collection("captains")
let $century := $x.century
group by $century
let $number := count($x)
let $number := count(distinct-values(for $series in $x.series
                                     return typeswitch($series)
                                            case array return $series()
                                            default return $series ))
where $number gt 1
return { "century" : $century, "number of series" : $number }
```

Result (run with Zorba):

{ "century" : 24, "number of series" : 3 }

5.6.6. Count clauses

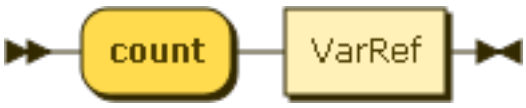

Figure 5.42. CountClause

For each incoming tuple, a binding from the position of this tuple in the tuple stream to the count variable is added. The new tuple is then forwarded to the next clause.

Example 5.120. An order by clause.

```
for $x in collection("captains")
order by $x.name
count $c
return { "id" : $c, "captain" : $x }
```

Result (run with Zorba):

{ "id" : 1, "captain" : { "name" : "Benjamin Sisko", "series" : ["The next generation", "Deep Space 9"], "century" : 24 } } { "id" : 2, "captain" : { "name" : "James T. Kirk", "series" : ["The original series"], "century" : 23 } } { "id" : 3, "captain" : { "name" : "Jean-Luc Picard", "series" : ["The next generation"], "century" : 24 } } { "id" : 4, "captain" : { "name" : "Jonathan Archer", "series" : ["Enterprise"], "century" : 22 } } { "id" : 5, "captain" : { "name" : "Kathryn Janeway", "series" : ["The next generation", "Voyager"], "century" : 24 } } { "id" : 6, "captain" : { "name" : "Samantha Carter", "series" : [ ], "century" : 21 } } { "id" : 7, "captain" : { "codename" : "Emergency Command Hologram", "surname" : "The Doctor", "series" : ["Voyager"], "century" : 24 } }

5.6.7. Map operator

Figure 5.43. SimpleMapExpr

Figure 5.44. ContextItemExpr

JSONiq provides a shortcut for a for-return construct, automatically binding each item in the left-hand-side sequence to the context item.

Example 5.121. A simple map

```
(1 to 10) ! ($$ * 2)
```

Result (run with Zorba):

2 4 6 8 10 12 14 16 18 20

Example 5.122. An equivalent query

```
for $i in 1 to 10
return $i * 2
```

Result (run with Zorba):

2 4 6 8 10 12 14 16 18 20

5.6.8. Composing FLWOR expressions

Like all other expressions, FLWOR expressions can be composed. In the following examples, a FLWOR is nested in a function call, nested in a FLWOR, nested in an array constructor:

Example 5.123. Nested FLWORs

```
[
  for $c in collection("captains")
  where exists(for $m in collection("movies")
               where some $moviecaptain in let $captain := $m.captain
                                           return typeswitch ($captain)
                                                  case array return $captain()
                                                  default return $captain
                     satisfies
                     $moviecaptain eq $c.name
               return $m)
  return $c.name
]
```

Result (run with Zorba):

["James T. Kirk", "Jean-Luc Picard"]

5.6.9. Ordered and Unordered expressions

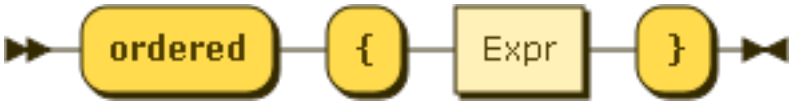

Figure 5.45. OrderedExpr

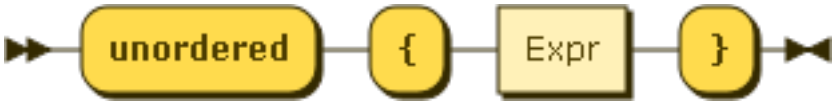

Figure 5.46. UnorderedExpr

By default, the order in which a for clause binds its items is important.

This behaviour can be relaxed in order give the optimizer more leeway. An unordered expression relaxes ordering by for clauses within its operand scope:

Example 5.124. An unordered expression.

```
unordered {
  for $captain in collection("captains")
  where $captain.century eq 24
  return $captain
}
```

Result (run with Zorba):

{ "name" : "Jean-Luc Picard", "series" : ["The next generation"], "century" : 24 } { "name" : "Benjamin Sisko", "series" : ["The next generation", "Deep Space 9"], "century" : 24 } { "name" : "Kathryn Janeway", "series" : ["The next generation", "Voyager"], "century" : 24 } { "codename" : "Emergency Command Hologram", "surname" : "The Doctor", "series" : ["Voyager"], "century" : 24 }

An ordered expression can be used to reactivate ordering behaviour in a subscope.

Example 5.125. An ordered expression.

```
unordered {
  for $captain in collection("captains")
  where ordered { exists(for $movie at $i in collection("movies")
                         where $i eq 5
                         where $movie.captain eq $captain.name
                         return $movie) }
  return $captain
}
```

Result (run with Zorba):

{ "name" : "James T. Kirk", "series" : ["The original series"], "century" : 23 }

5.7. Expressions dealing with types

This section describes JSONiq types as well as the sequence type syntax.

5.7.1. Instance-of expressions

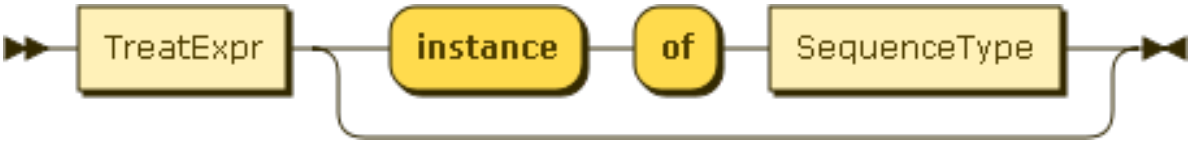

Figure 5.47. InstanceofExpr

An instance expression can be used to tell whether a JSONiq value matches a given sequence type.

Example 5.126. Instance of expression

```
1 instance of integer
```

Result (run with Zorba):

true

Example 5.127. Instance of expression

```
1 instance of string
```

Result (run with Zorba):

false

Example 5.128. Instance of expression

```
"foo" instance of string
```

Result (run with Zorba):

true

Example 5.129. Instance of expression

```
{ "foo" : "bar" } instance of object
```

Result (run with Zorba):

true

Example 5.130. Instance of expression

```
({ "foo" : "bar" }, { "bar" : "foo" }) instance of json-item+
```

Result (run with Zorba):

true

Example 5.131. Instance of expression

```
[ 1, 2, 3 ] instance of array?
```

Result (run with Zorba):

true

Example 5.132. Instance of expression

```
() instance of ()
```

Result (run with Zorba):

true

5.7.2. Treat expressions

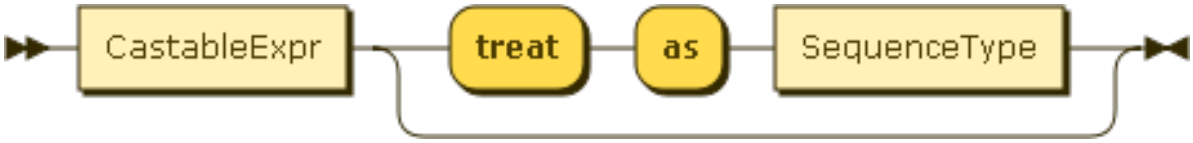

Figure 5.48. TreatExpr

A treat expression checks that a JSONiq value matches a given sequence type. If it is not the case, an error is raised.

Example 5.133. Treat as expression

```
1 treat as integer
```

Result (run with Zorba):

1

Example 5.134. Treat as expression

```
1 treat as string
```

Result (run with Zorba):

An error was raised: "xs:integer" cannot be treated as type xs:string

Example 5.135. Treat as expression

```
"foo" treat as string
```

Result (run with Zorba):

foo

Example 5.136. Treat as expression

```
{ "foo" : "bar" } treat as object
```

Result (run with Zorba):

{ "foo" : "bar" }

Example 5.137. Treat as expression

```
({ "foo" : "bar" }, { "bar" : "foo" }) treat as json-item+
```

Result (run with Zorba):

{ "foo" : "bar" } { "bar" : "foo" }

Example 5.138. Treat as expression

```
[ 1, 2, 3 ] treat as array?
```

Result (run with Zorba):

[1, 2, 3]

Example 5.139. Treat as expression

```
() treat as ()
```

Result (run with Zorba):

5.7.3. Castable expressions

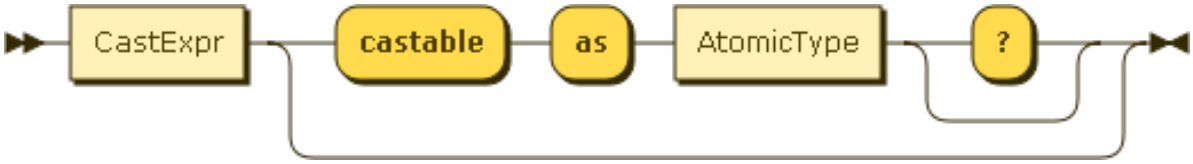

Figure 5.49. CastableExpr

A castable expression checks whether a JSONiq value can be cast to a given atomic type and returns true or false accordingly. It can be used before actually casting to that type.

Example 5.140. Castable as expression

```
"1" castable as integer
```

Result (run with Zorba):

true

Example 5.141. Castable as expression

```
"foo" castable as integer
```

Result (run with Zorba):

false

Example 5.142. Castable as expression

```
"2013-04-02" castable as date
```

Result (run with Zorba):

true

Example 5.143. Castable as expression

```
() castable as date
```

Result (run with Zorba):

false

Example 5.144. Castable as expression

```
("2013-04-02", "2013-04-03") castable as date
```

Result (run with Zorba):

false

The question mark allows for an empty sequence.

Example 5.145. Castable as expression

```
() castable as date?
```

Result (run with Zorba):

true

5.7.4. Cast expressions

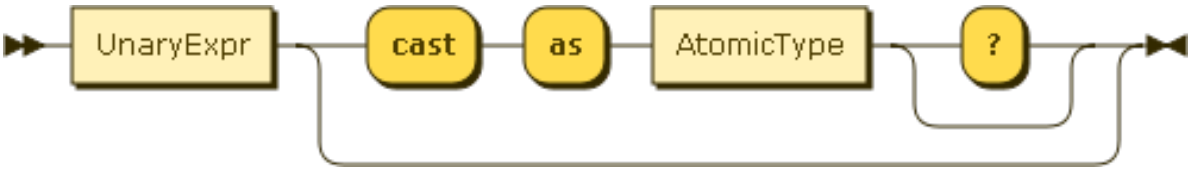

Figure 5.50. CastExpr

A cast expression casts a JSONiq value to a given atomic type. The resulting value is annotated with this type.

Example 5.146. Cast as expression

```
"1" cast as integer
```

Result (run with Zorba):

1

Example 5.147. Cast as expression

```
"foo" cast as integer
```

Result (run with Zorba):

An error was raised: "foo": value of type xs:string is not castable to type xs:integer

Example 5.148. Cast as expression

```
"2013-04-02" cast as date
```

Result (run with Zorba):

2013-04-02

Example 5.149. Cast as expression

```
() cast as date
```

Result (run with Zorba):

An error was raised: empty sequence can not be cast to type with quantifier '1'

Example 5.150. Cast as expression

```
("2013-04-02", "2013-04-03") cast as date
```

Result (run with Zorba):

An error was raised: sequence of more than one item can not be cast to type with quantifier '1' or '?'

The question mark allows for an empty sequence.

Example 5.151. Cast as expression

```
() cast as date?
```

Result (run with Zorba):

Example 5.152. Cast as expression

```
"2013-04-02" cast as date?
```

Result (run with Zorba):

2013-04-02

5.7.5. Typeswitch expressions

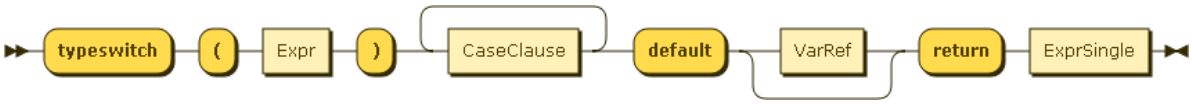

Figure 5.51. TypeswitchExpr

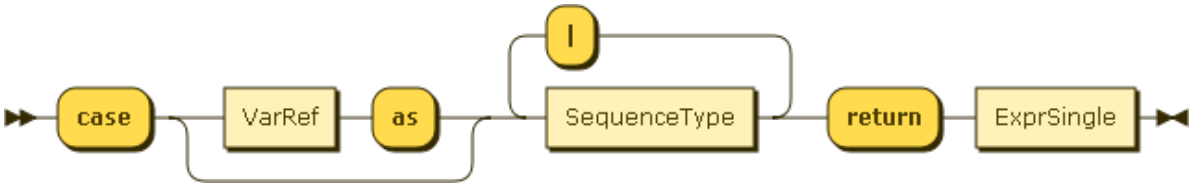

Figure 5.52. CaseClause

A typeswitch expressions tests if the value resulting from the first operand matches a given list of types. The expression corresponding to the first matching case is finally evaluated. If there is no match, the expression in the default clause is evaluated.

Example 5.153. Typeswitch expression

```
typeswitch("foo")
case integer return "integer"
case string return "string"
case object return "object"
default return "other"
```

Result (run with Zorba):

string

In each clause, it is possible to bind the value of the first operand to a variable.

Example 5.154. Typeswitch expression

```
typeswitch("foo")
case $i as integer return $i + 1
case $s as string return $s || "foo"
case $o as object return [ $o ]
default $d return $d
```

Result (run with Zorba):

foofoo

The vertical bar can be used to allow several types in the same case clause.

Example 5.155. Typeswitch expression

```
typeswitch("foo")
case $a as integer | string return { "integer or string" : $a }
case $o as object return [ $o ]
default $d return $d
```

Result (run with Zorba):

{ "integer or string" : "foo" }

Prologs

This section introduces prologs, which allows declaring functions and global variables that can then be used in the main query. A prolog also allows setting some default behaviour.

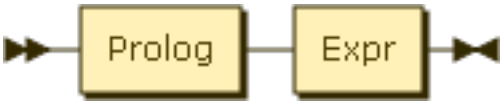

Figure 6.1. MainModule

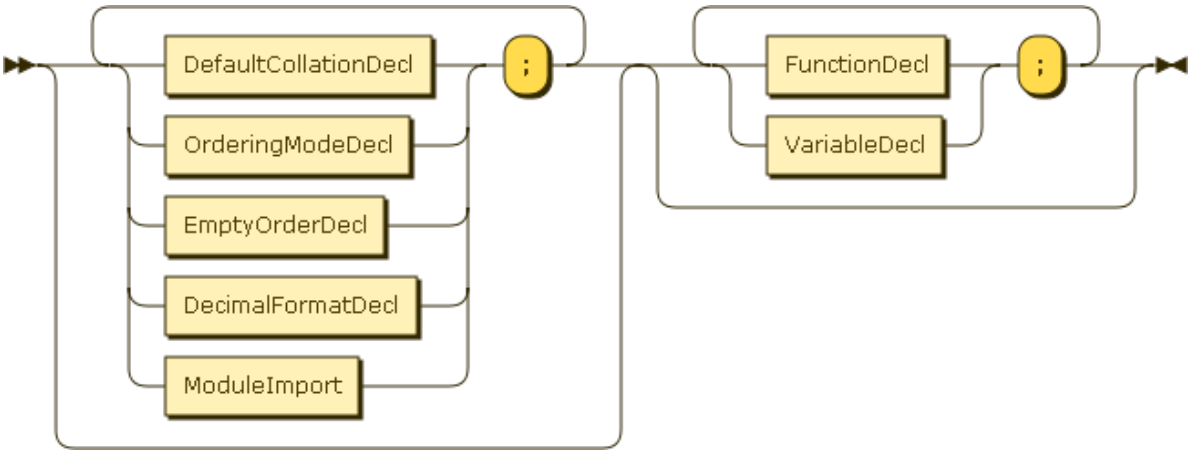

Figure 6.2. Prolog

The prolog appears before the main query and is optional. It can contain setters and module imports, followed by function and variable declarations.

Module imports are explained in the next chapter.

6.1. Setters.

Figure 6.3. Setter

Setters allow to specify a default behaviour for various aspects of the language.

6.1.1. Default collation

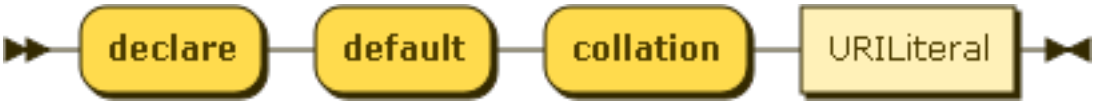

Figure 6.4. DefaultCollationDecl

This specifies the default collation used for grouping and ordering clauses in FLWOR expressions. It can be overriden with a collation directive in these clauses.

6.1.2. Default ordering mode

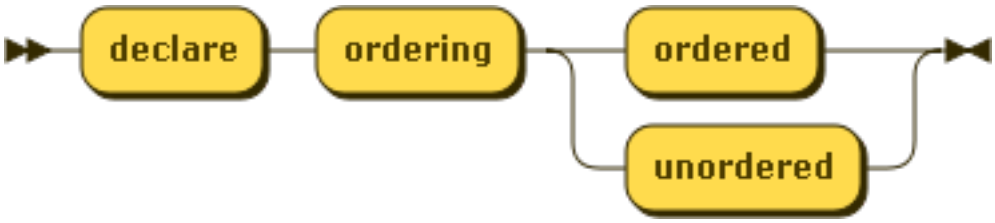

Figure 6.5. OrderingModeDecl

This specifies the default behaviour of from clauses, i.e., if they bind tuples in the order in which items occur in the binding sequence. It can be overriden with ordered and unordered expressions.

6.1.3. Default ordering behaviour for empty sequences

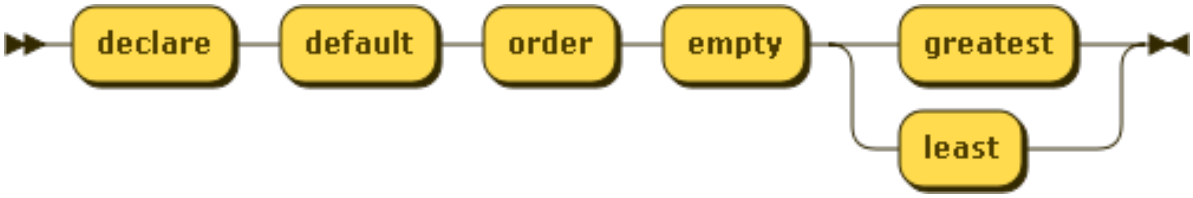

Figure 6.6. EmptyOrderDecl

This specifies whether empty sequences come first or last in an ordering clause. It can be overriden by the corresponding directives in such clauses.

6.1.4. Default decimal format

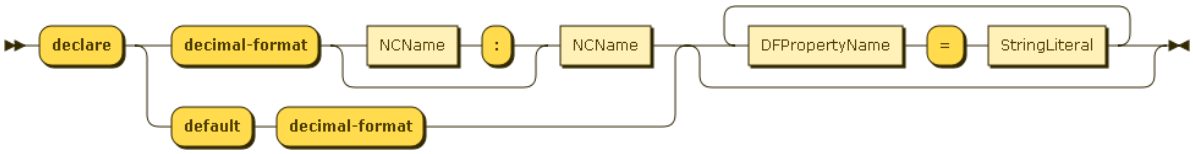

Figure 6.7. DecimalFormatDecl

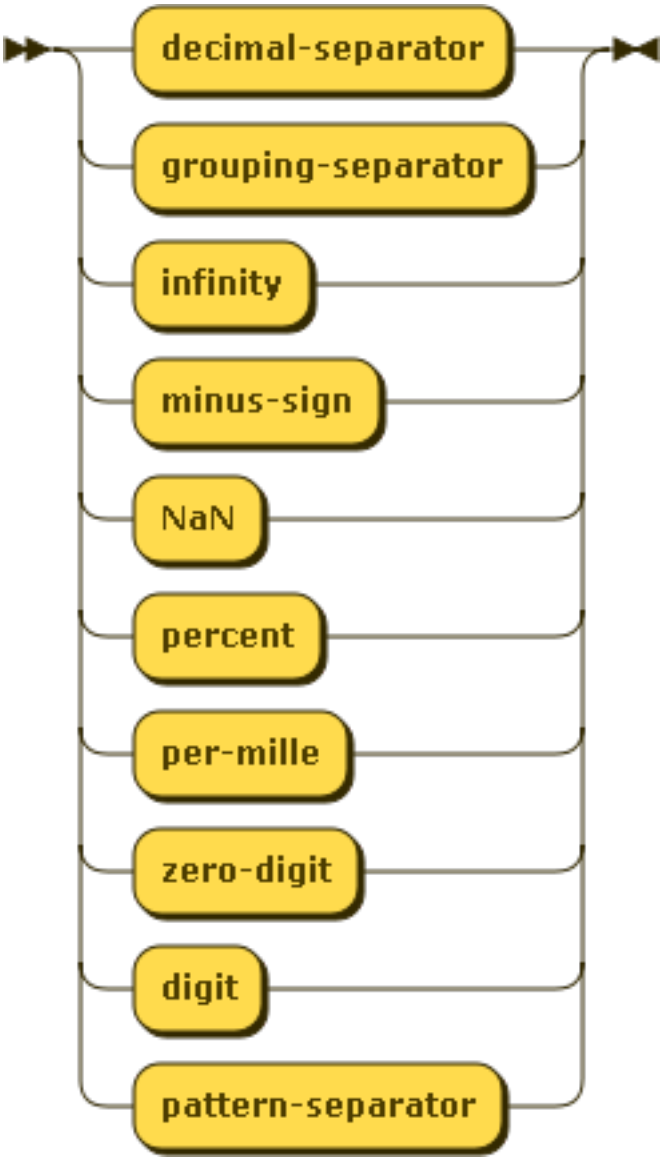

Figure 6.8. DFPropertyName

This specifies a default decimal format for the builtin function format-number().

6.2. Global variables

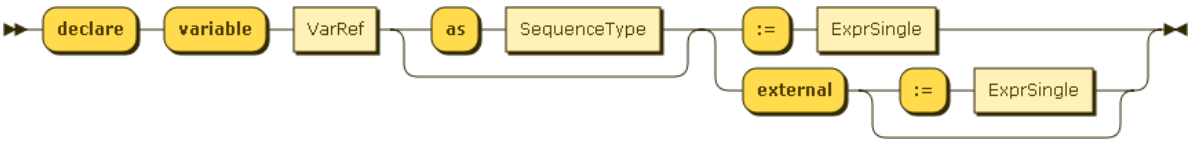

Figure 6.9. VarDecl

Variables can be declared global. Global variables are declared in the prolog.

> Example 6.1. Global variable
>
> ```
> declare variable $obj := { "foo" : "bar" };
> $obj
> ```

Result (run with Zorba):

{ "foo" : "bar" }

Example 6.2. Global variable

```
declare variable $numbers := (1, 2, 3, 4, 5);
[ $numbers ]
```

Result (run with Zorba):

[1, 2, 3, 4, 5]

You can specify a type for a variable. If the type does not match, an error is raised. Types will be explained later. In general, you do not need to worry too much about variable types except if you want to make sure that what you bind to a variable is really what you want. In most cases, the engine will take care of types for you.

Example 6.3. Global variable with a type

```
declare variable $obj as object := { "foo" : "bar" };
$obj
```

Result (run with Zorba):

{ "foo" : "bar" }

An external variable allows you to pass a value from the outside environment, which can be very useful. Each implementation can choose their own way of passing a value to an external variable. A default value for an external variable can also be supplied in case none is provided outside.

Example 6.4. An external global variable

```
declare variable $obj external;
$obj
```

Result (run with Zorba):

An error was raised: "obj": variable has no value

Example 6.5. An external global variable with a default value

```
declare variable $obj external := { "foo" : "bar" };
$obj
```

Result (run with Zorba):

{ "foo" : "bar" }

6.3. Functions

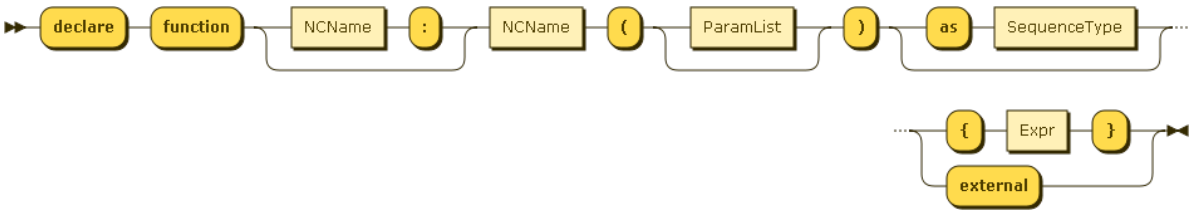

Figure 6.10. VarDecl

You can define your own functions in the prolog. These user-defined functions must be prefixed with *local:*, both in the declaration and when called.

Remember than types are optional, and if you do not specify any, item* is assumed, both for parameters and for the return type.

Example 6.6. An external global variable with a default value

```
declare function local:say-hello($x) { "Hello, " || $x || "!" };
local:say-hello("Mister Spock")
```

Result (run with Zorba):

Hello, Mister Spock!

Example 6.7. An external global variable with a default value

```
declare function local:say-hello($x as string) { "Hello, " || $x || "!" };
local:say-hello("Mister Spock")
```

Result (run with Zorba):

Hello, Mister Spock!

Example 6.8. An external global variable with a default value

```
declare function local:say-hello($x as string) as string { "Hello, " || $x || "!" };
local:say-hello("Mister Spock")
```

Result (run with Zorba):

Hello, Mister Spock!

If you do specify types, an error is raised in case of a mismatch

Example 6.9. An external global variable with a default value

```
declare function local:say-hello($x) { "Hello, " || $x || "!" };
```

```
local:say-hello(1)
```

Result (run with Zorba):

Hello, 1!

Modules

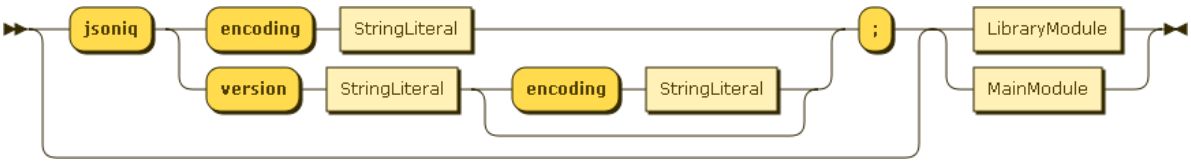

Figure 7.1. Module

You can group functions and variables in separate library modules.

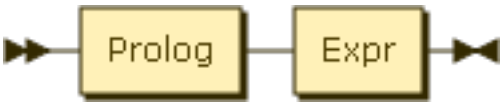

Figure 7.2. MainModule

Up to now, everything we encountered were main modules, i.e., a prolog followed by a main query.

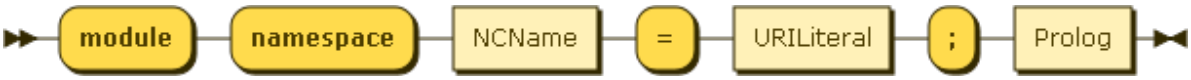

Figure 7.3. LibraryModule

Figure 7.4. ModuleDecl

A library module does not contain any query - just functions and variables that can be imported by other modules.

A library module must be assigned to a namespace. For convenience, this namespace is bound to an alias in the module declaration. All variables and functions in a library module must be prefixed with this alias.

Example 7.1. A library module

```
module namespace my = "http://www.example.com/my-module";
declare variable $my:variable := { "foo" : "bar" };
declare variable $my:n := 42;
declare function my:function($i as integer) { $i * $i };
```

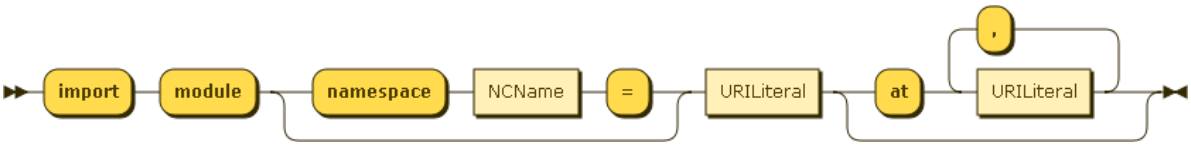

Figure 7.5. ModuleImport

Here is a main module which imports the former library module. An alias is given to the module namespace (my). Variables and functions from that module can be accessed by prefixing their names with this alias. The alias may be different than the internal alias defined in the imported module.

Example 7.2. An importing main module

```
import module namespace other= "http://www.example.com/my-module";
other:function($other:n)
```

Result (run with Zorba):

1764

Function Library

JSONiq provides a rich set of functions.

8.1. JSON specific functions.

Some functions are specific to JSON.

8.1.1. keys

This function returns the distinct keys of all objects in the supplied sequence, in an implementation-dependent order.

`keys($o as item*) as string*`

Example 8.1. Getting all distinct key names in the supplied objects, ignoring non-objects.

```
let $o := ("foo", [ 1, 2, 3 ], { "a" : 1, "b" : 2 }, { "a" : 3, "c" : 4 })
return keys($o)
```

Result (run with Zorba):

a b c

Example 8.2. Retrieving all Pairs from an Object:

```
let $map := { "eyes" : "blue", "hair" : "fuchsia" }
for $key in keys($map)
return { $key : $map.$key }
```

Result (run with Zorba):

{ "eyes" : "blue" } { "hair" : "fuchsia" }

8.1.2. members

This functions returns all members of all arrays of the supplied sequence.

`members($a as item*) as item*`

Example 8.3. Retrieving the members of all supplied arrays, ignoring non-arrays.

```
let $planets :=  ( "foo", { "foo" : "bar "}, [ "mercury", "venus", "earth", "mars" ], [ 1,
 2, 3 ])
return members($planets)
```

Result (run with Zorba):

mercury venus earth mars 1 2 3

8.1.3. null

This function returns the JSON null.

`null() as null`

8.1.4. parse-json

This function parses its first parameter (a string) as JSON, and returns the resulting sequence of objects and arrays.

`parse-json($arg as string?) as json-item*`

`parse-json($arg as string?, $options as object) as json-item*`

The object optionally supplied as the second parameter may contain additional options:

- **`jsoniq-multiple-top-level-items`** (boolean): indicates whether parsing to zero, or several objects is allowed. An error is raised if this value is false and there is not exactly one object that was parsed.

If parsing is not successful, an error is raised. Parsing is considered in particular to be non-successful if the boolean associated with "jsoniq-multiple-top-level-items" in the additional parameters is false and there is extra content after parsing a single abject or array.

Example 8.4. Parsing a JSON document

```
parse-json("{ \"foo\" : \"bar\" }", { "jsoniq-multiple-top-level-items" : false })
```

Result (run with Zorba):

{ "foo" : "bar" }

Example 8.5. Parsing multiple, whitespace-separated JSON documents

```
parse-json("{ \"foo\" : \"bar\" } { \"bar\" : \"foo\" }")
```

Result (run with Zorba):

{ "foo" : "bar" } { "bar" : "foo" }

8.1.5. size

This function returns the size of the supplied array, or the empty sequence if the empty sequence is provided.

`size($a as array?) as integer?`

Example 8.6. Retrieving the size of an array

```
let $a := [1 to 10]
return size($a)
```

Result (run with Zorba):

10

8.1.6. accumulate

This function dynamically builds an object, like the {| |} syntax, except that it does not throw an error upon pair collision. Instead, it accumulates them, wrapping into an array if necessary. Non-objects are ignored.

```
declare function accumulate($seq as item*) as object
{
  {|
    keys($seq) ! { $$ : $seq.$$ }
  |}
};
```

8.1.7. descendant-arrays

This function returns all arrays contained within the supplied items, regardless of depth.

```
declare function descendant-arrays($seq as item*) as array*
{
  for $i in $seq
  return typeswitch ($i)
  case array return ($i, descendant-arrays($i[])
  case object return return descendant-arrays(values($i))
  default return ()
};
```

8.1.8. descendant-objects

This function returns all objects contained within the supplied items, regardless of depth.

```
declare function descendant-objects($seq as item*) as object*
{
  for $i in $seq
  return typeswitch ($i)
  case object return ($i, descendant-objects(values($i)))
  case array return return descendant-objects($i[])
  default return ()
};
```

8.1.9. descendant-pairs

This function returns all descendant pairs within the supplied items.

```
declare function descendant-pairs($seq as item*)
{
  for $i in $seq
  return typeswitch ($i)
  case object return
    for $k in keys($o)
    let $v := $o.$k
    return ({ $k : $v }, descendant-pairs($v))
  case array return descendant-pairs($i[])
```

```
  default return ()
};
```

Example 8.7. Accessing all descendant pairs

```
let $o :=
{
  "first" : 1,
  "second" : {
    "first" : "a",
    "second" : "b"
  }
}
return descendant-pairs($o)
```

Result (run with Zorba):

An error was raised: "descendant-pairs": function with arity 1 not declared

8.1.10. flatten

This function recursively flattens arrays in the input sequence, leaving non-arrays intact.

```
declare function flatten($seq as item*) as item*
{
  for $value in $seq
  return typeswitch ($value)
         case array return flatten($value[])
         default return $value
};
```

8.1.11. intersect

This function returns the intersection of the supplied objects, and aggregates values corresponding to the same name into an array. Non-objects are ignored.

```
declare function intersect($seq as item*)
{
  {|
    let $objects := $seq[. instance of object()]
    for $key in keys(head($objects))
    where every $object in tail($objects)
          satisfies exists(index-of(keys($object), $key))
    return { $key : $objects.$key }
  |}
};
```

8.1.12. project

This function iterates on the input sequence. It projects objects by filtering their pairs and leaves non-objects intact.

```
declare function project($seq as item*, $keys as string*) as item*
{
```

```
  for $item in $seq
  return typeswitch ($item)
         case $object as object return
         {|
           for $key in keys($object)
           where some $to-project in $keys satisfies $to-project eq $key
           let $value := $object.$key
           return { $key : $value }
         |}
         default return $item
};
```

Example 8.8. Projecting an object 1

```
let $o := {
  "Captain" : "Kirk",
  "First Officer" : "Spock",
  "Engineer" : "Scott"
  }
return project($o, ("Captain", "First Officer"))
```

Result (run with Zorba):

{ "Captain" : "Kirk", "First Officer" : "Spock" }

Example 8.9. Projecting an object 2

```
let $o := {
  "Captain" : "Kirk",
  "First Officer" : "Spock",
  "Engineer" : "Scott"
  }
return project($o, "XQuery Evangelist")
```

Result (run with Zorba):

{ }

8.1.13. remove-keys

This function iterates on the input sequence. It removes the pairs with the given keys from all objects and leaves non-objects intact.

```
declare function remove-keys($seq as item*, $keys as string*) as item*
{
  for $item in $seq
  return typeswitch ($item)
         case $object as object return
         {|
           for $key in keys($object)
           where every $to-remove in $keys satisfies $to-remove ne $key
           let $value := $object.$key
           return { $key : $value }
         |}
         default return $item
};
```

Example 8.10. Removing keys from an object (not implemented yet)

```
let $o := {
  "Captain" : "Kirk",
  "First Officer" : "Spock",
  "Engineer" : "Scott"
  }
return remove-keys($o, ("Captain", "First Officer"))
```

Result (run with Zorba):

An error was raised: "remove-keys": function with arity 2 not declared

8.1.14. values

This function returns all values in the supplied objects. Non-objects are ignored.

```
declare function values($seq as item*) as item* {
  for $i in $seq
  for $k in jn:keys($i)
  return $i($k)
};
```

8.1.15. encode-for-roundtrip

This function encodes any sequence of items, even containing non-JSON types, to a sequence of JSON items that can be serialized as pure JSON, in a way that it can be parsed and decoded back using decode-from-roundtrip. JSON features are left intact, while atomic items annotated with a non-JSON type are converted to objects embedding all necessary information.

`encode-for-roundtrip($items as item*) as json-item*`

8.1.16. decode-from-roundtrip

This function decodes a sequence previously encoded with encode-for-roundtrip.

`decode-from-roundtrip($items as json-item*) as item*`

8.2. Functions taken from XQuery

- Access to the external environment: *collection#1*[1]
- Function to turn atomics into booleans for use in two-valued logics: *boolean#1*[2]
- Raising errors: *error#0*[3], *error#1*[4], *error#2*[5], *error#3*[6].

[1] http://www.zorba-xquery.com/html/modules/w3c/xpath#collection-1
[2] http://www.zorba-xquery.com/html/modules/w3c/xpath#boolean-1
[3] http://www.zorba-xquery.com/html/modules/w3c/xpath#error-0
[4] http://www.zorba-xquery.com/html/modules/w3c/xpath#error-1
[5] http://www.zorba-xquery.com/html/modules/w3c/xpath#error-2
[6] http://www.zorba-xquery.com/html/modules/w3c/xpath#error-3

- Functions on numeric values: *abs#1*[7], *ceilingabs#1*[8], *floorabs#1*[9], *roundabs#1*[10], *round-half-to-even#1*[11]
- Parsing numbers: *number#0*[12], *number#1*[13]
- Formatting integers: *format-integer#2*[14], *format-integer#3*[15]
- Formatting numbers: *format-number#2*[16], *format-number#3*[17]
- Trigonometric and exponential functions: *pi#0*[18], *exp#1*[19], *exp10#1*[20], *log#1*[21], *log10#1*[22], *pow#2*[23], *sqrt#1*[24], *sin#1*[25], *cos#1*[26], *tan#1*[27], *asin#1*[28], *acos#1*[29], *atan#1*[30], *atan2#1*[31]
- Functions to assemble and disassemble strings: *codepoints-to-string#1*[32], *string-to-codepoints#1*[33]
- Comparison of strings: *compare#2*[34], *compare#3*[35], *codepoint-equal#2*[36]
- Functions on string values: *concat#2*[37], *string-join#1*[38], *string-join#2*[39], *substring#2*[40], *substring#3*[41], *string-length#0*[42], *string-length#1*[43], *normalize-space#0*[44], *normalize-space#1*[45], *normalize-unicode#1*[46], *normalize-unicode#2*[47], *upper-case#1*[48], *lower-case#1*[49], *translate#3*[50]

[7] http://www.zorba-xquery.com/html/modules/w3c/xpath#abs-1
[8] http://www.zorba-xquery.com/html/modules/w3c/xpath#ceilingabs-1
[9] http://www.zorba-xquery.com/html/modules/w3c/xpath#floorabs-1
[10] http://www.zorba-xquery.com/html/modules/w3c/xpath#roundabs-1
[11] http://www.zorba-xquery.com/html/modules/w3c/xpath#round-half-to-even-1
[12] http://www.zorba-xquery.com/html/modules/w3c/xpath#number-0
[13] http://www.zorba-xquery.com/html/modules/w3c/xpath#number-1
[14] http://www.zorba-xquery.com/html/modules/w3c/xpath#format-integer-2
[15] http://www.zorba-xquery.com/html/modules/w3c/xpath#format-integer-3
[16] http://www.zorba-xquery.com/html/modules/w3c/xpath#format-number-2
[17] http://www.zorba-xquery.com/html/modules/w3c/xpath#format-number-3
[18] http://www.zorba-xquery.com/html/modules/w3c/xpath#pi-0
[19] http://www.zorba-xquery.com/html/modules/w3c/xpath#exp-1
[20] http://www.zorba-xquery.com/html/modules/w3c/xpath#exp10-1
[21] http://www.zorba-xquery.com/html/modules/w3c/xpath#log-1
[22] http://www.zorba-xquery.com/html/modules/w3c/xpath#log10-1
[23] http://www.zorba-xquery.com/html/modules/w3c/xpath#pow-2
[24] http://www.zorba-xquery.com/html/modules/w3c/xpath#sqrt-1
[25] http://www.zorba-xquery.com/html/modules/w3c/xpath#sin-1
[26] http://www.zorba-xquery.com/html/modules/w3c/xpath#cos-1
[27] http://www.zorba-xquery.com/html/modules/w3c/xpath#tan-1
[28] http://www.zorba-xquery.com/html/modules/w3c/xpath#asin-1
[29] http://www.zorba-xquery.com/html/modules/w3c/xpath#acos-1
[30] http://www.zorba-xquery.com/html/modules/w3c/xpath#atan-1
[31] http://www.zorba-xquery.com/html/modules/w3c/xpath#atan2-1
[32] http://www.zorba-xquery.com/html/modules/w3c/xpath#codepoints-to-string-1
[33] http://www.zorba-xquery.com/html/modules/w3c/xpath#string-to-codepoints-1
[34] http://www.zorba-xquery.com/html/modules/w3c/xpath#compare-2
[35] http://www.zorba-xquery.com/html/modules/w3c/xpath#compare-3
[36] http://www.zorba-xquery.com/html/modules/w3c/xpath#codepoint-equal-2
[37] http://www.zorba-xquery.com/html/modules/w3c/xpath#concat-2
[38] http://www.zorba-xquery.com/html/modules/w3c/xpath#string-join-1
[39] http://www.zorba-xquery.com/html/modules/w3c/xpath#string-join-2
[40] http://www.zorba-xquery.com/html/modules/w3c/xpath#substring-2
[41] http://www.zorba-xquery.com/html/modules/w3c/xpath#substring-3
[42] http://www.zorba-xquery.com/html/modules/w3c/xpath#string-length-0
[43] http://www.zorba-xquery.com/html/modules/w3c/xpath#string-length-1
[44] http://www.zorba-xquery.com/html/modules/w3c/xpath#normalize-space-0
[45] http://www.zorba-xquery.com/html/modules/w3c/xpath#normalize-space-1
[46] http://www.zorba-xquery.com/html/modules/w3c/xpath#normalize-unicode-1
[47] http://www.zorba-xquery.com/html/modules/w3c/xpath#normalize-unicode-2
[48] http://www.zorba-xquery.com/html/modules/w3c/xpath#upper-case-1

- Functions based on substring matching: *contains#2*[51], *contains#3*[52], *starts-with#2*[53], *starts-with#3*[54], *ends-with#2*[55], *ends-with#3*[56], *substring-before#2*[57], *substring-before#3*[58], *substring-after#2*[59], *substring-after#3*[60]
- String functions that use regular expressions: *matches#2*[61], *matches#3*[62], *replace#3*[63], *replace#4*[64], *tokenize#2*[65], *tokenize#3*[66]
- Functions that manipulate URIs: *resolve-uri#1*[67], *resolve-uri#2*[68], *encode-for-uri#1*[69], *iri-to-uri#1*[70], *escape-html-uri#1*[71]
- General functions on sequences: *empty#1*[72], *exists#1*[73], *head#1*[74], *tail#1*[75], *insert-before#3*[76], *remove#2*[77], *reverse#1*[78], *subsequence#2*[79], *subsequence#3*[80], *unordered#1*[81]
- Function that compare values in sequences: *distinct-values#1*[82], *distinct-values#2*[83], *index-of#2*[84], *index-of#3*[85], *deep-equal#2*[86]. *deep-equal#3*[87]
- Functions that test the cardinality of sequences: *zero-or-one#1*[88], *one-or-more#1*[89], *exactly-one#1*[90]

[49] http://www.zorba-xquery.com/html/modules/w3c/xpath#lower-case-1
[50] http://www.zorba-xquery.com/html/modules/w3c/xpath#translate-3
[51] http://www.zorba-xquery.com/html/modules/w3c/xpath#contains-2
[52] http://www.zorba-xquery.com/html/modules/w3c/xpath#contains-3
[53] http://www.zorba-xquery.com/html/modules/w3c/xpath#starts-with-2
[54] http://www.zorba-xquery.com/html/modules/w3c/xpath#starts-with-3
[55] http://www.zorba-xquery.com/html/modules/w3c/xpath#ends-with-2
[56] http://www.zorba-xquery.com/html/modules/w3c/xpath#ends-with-3
[57] http://www.zorba-xquery.com/html/modules/w3c/xpath#substring-before-2
[58] http://www.zorba-xquery.com/html/modules/w3c/xpath#substring-before-3
[59] http://www.zorba-xquery.com/html/modules/w3c/xpath#substring-after-2
[60] http://www.zorba-xquery.com/html/modules/w3c/xpath#substring-after-3
[61] http://www.zorba-xquery.com/html/modules/w3c/xpath#matches-2
[62] http://www.zorba-xquery.com/html/modules/w3c/xpath#matches-3
[63] http://www.zorba-xquery.com/html/modules/w3c/xpath#replace-3
[64] http://www.zorba-xquery.com/html/modules/w3c/xpath#replace-4
[65] http://www.zorba-xquery.com/html/modules/w3c/xpath#tokenize-2
[66] http://www.zorba-xquery.com/html/modules/w3c/xpath#tokenize-3
[67] http://www.zorba-xquery.com/html/modules/w3c/xpath#resolve-uri-1
[68] http://www.zorba-xquery.com/html/modules/w3c/xpath#resolve-uri-2
[69] http://www.zorba-xquery.com/html/modules/w3c/xpath#encode-for-uri-1
[70] http://www.zorba-xquery.com/html/modules/w3c/xpath#iri-to-uri-1
[71] http://www.zorba-xquery.com/html/modules/w3c/xpath#escape-html-uri-1
[72] http://www.zorba-xquery.com/html/modules/w3c/xpath#empty-1
[73] http://www.zorba-xquery.com/html/modules/w3c/xpath#exists-1
[74] http://www.zorba-xquery.com/html/modules/w3c/xpath#head-1
[75] http://www.zorba-xquery.com/html/modules/w3c/xpath#tail-1
[76] http://www.zorba-xquery.com/html/modules/w3c/xpath#insert-before-3
[77] http://www.zorba-xquery.com/html/modules/w3c/xpath#remove-2
[78] http://www.zorba-xquery.com/html/modules/w3c/xpath#reverse-1
[79] http://www.zorba-xquery.com/html/modules/w3c/xpath#subsequence-2
[80] http://www.zorba-xquery.com/html/modules/w3c/xpath#subsequence-3
[81] http://www.zorba-xquery.com/html/modules/w3c/xpath#unordered-1
[82] http://www.zorba-xquery.com/html/modules/w3c/xpath#distinct-values-1
[83] http://www.zorba-xquery.com/html/modules/w3c/xpath#distinct-values-2
[84] http://www.zorba-xquery.com/html/modules/w3c/xpath#index-of-2
[85] http://www.zorba-xquery.com/html/modules/w3c/xpath#index-of-3
[86] http://www.zorba-xquery.com/html/modules/w3c/xpath#deep-equal-2
[87] http://www.zorba-xquery.com/html/modules/w3c/xpath#deep-equal-3
[88] http://www.zorba-xquery.com/html/modules/w3c/xpath#zero-or-one-1
[89] http://www.zorba-xquery.com/html/modules/w3c/xpath#one-or-more-1
[90] http://www.zorba-xquery.com/html/modules/w3c/xpath#exactly-one-1

- Aggregate functions: *count#1*[91], *avg#1*[92], *max#1*[93], *min#1*[94], *sum#1*[95]
- Serializing functions: *serialize#1*[96] (unary)
- Context information: *current-dateTime#1*[97], *current-date#1*[98], *current-time#1*[99], *implicit-time-zone#1*[100], *default-collation#1*[101]
- Constructor functions: for all builtin types, with the name of the builtin type and unary. Equivalent to a cast expression.

[91] http://www.zorba-xquery.com/html/modules/w3c/xpath#count-1
[92] http://www.zorba-xquery.com/html/modules/w3c/xpath#avg-1
[93] http://www.zorba-xquery.com/html/modules/w3c/xpath#max-1
[94] http://www.zorba-xquery.com/html/modules/w3c/xpath#min-1
[95] http://www.zorba-xquery.com/html/modules/w3c/xpath#sum-1
[96] http://www.zorba-xquery.com/html/modules/w3c/xpath#serialize-1
[97] http://www.zorba-xquery.com/html/modules/w3c/xpath#current-dateTime-1
[98] http://www.zorba-xquery.com/html/modules/w3c/xpath#current-date-1
[99] http://www.zorba-xquery.com/html/modules/w3c/xpath#current-time-1
[100] http://www.zorba-xquery.com/html/modules/w3c/xpath#implicit-timezone-1
[101] http://www.zorba-xquery.com/html/modules/w3c/xpath#default-collation-1

Equality and identity

As in most language, one can distinguish between physical equality and logical equality.

Atomics can only be compared logically. Their physically identity is totally opaque to you.

Example 9.1. Logical comparison of two atomics

```
1 eq 1
```

Result (run with Zorba):

true

Example 9.2. Logical comparison of two atomics

```
1 eq 2
```

Result (run with Zorba):

false

Example 9.3. Logical comparison of two atomics

```
"foo" eq "bar"
```

Result (run with Zorba):

false

Example 9.4. Logical comparison of two atomics

```
"foo" ne "bar"
```

Result (run with Zorba):

true

Two objects or arrays can be tested for logical equality as well, using deep-equal(), which performs a recursive comparison.

Example 9.5. Logical comparison of two JSON items

```
deep-equal({ "foo" : "bar" }, { "foo" : "bar" })
```

Result (run with Zorba):

true

Example 9.6. Logical comparison of two JSON items

```
deep-equal({ "foo" : "bar" }, { "bar" : "foo" })
```

Result (run with Zorba):

false

The physical identity of objects and arrays is not exposed to the user in the core JSONiq language itself. Some library modules might be able to reveal it, though.

Notes

10.1. Sequences vs. Arrays

Even though JSON supports arrays, JSONiq uses a different construct as its first class citizens: sequences. Any value returned by or passed to an expression is a sequence.

The main difference between sequences and arrays is that sequences are completely flat, meaning they cannot contain other sequences.

Since sequences are flat, expressions of the JSONiq language just concatenate them to form bigger sequences.

This is crucial to allow streaming results, for example through an HTTP session.

> Example 10.1. Flat sequences
>
> ```
> ( (1, 2), (3, 4) )
> ```
>
> **Result (run with Zorba):**
> 1 2 3 4

Arrays on the other side can contain nested arrays, like in JSON.

> Example 10.2. Nesting arrays
>
> ```
> [ [ 1, 2 ], [ 3, 4 ] ]
> ```
>
> **Result (run with Zorba):**
> [[1, 2], [3, 4]]

Many expressions return single items - actually, they really return a singleton sequence, but a singleton sequence of one item is considered the same as this item.

> Example 10.3. Singleton sequences
>
> ```
> 1 + 1
> ```
>
> **Result (run with Zorba):**
> 2

This is different for arrays: a singleton array is distinct from its unique member, like in JSON.

> Example 10.4. Singleton sequences
>
> ```
> ```

```
[ 1 + 1 ]
```

Result (run with Zorba):

[2]

An array is a single item. A (non-singleton) sequence is not. This can be observed by counting the number of items in a sequence.

Example 10.5. count() on an array

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

Result (run with Zorba):

1

Example 10.6. count() on a sequence

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

Result (run with Zorba):

4

Other than that, arrays and sequences can contain exactly the same members (atomics, arrays, objects).

Example 10.7. Members of an array

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

Result (run with Zorba):

[1, "foo", [1, 2, 3, 4], { "foo" : "bar" }]

Example 10.8. Members of an sequence

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

Result (run with Zorba):

1 foo [1, 2, 3, 4] { "foo" : "bar" }

Arrays can be converted to sequences, and vice-versa.

Example 10.9. Converting an array to a sequence

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

Result (run with Zorba):

1 foo [1, 2, 3, 4] { "foo" : "bar" }

Example 10.10. Converting a sequence to an array

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

Result (run with Zorba):

[1, "foo", [1, 2, 3, 4], { "foo" : "bar" }]

10.2. Null vs. empty sequence

Null and the empty sequence are two different concepts.

Null is an item (an atomic value), and can be a member of an array or of a sequence, or the value associated with a key in an object. Sequences cannot, as they represent the absence of any item.

Example 10.11. Null values in an array

```
[ null, 1, null, 2 ]
```

Result (run with Zorba):

[null, 1, null, 2]

Example 10.12. Null values in an object

```
{ "foo" : null }
```

Result (run with Zorba):

{ "foo" : null }

Example 10.13. Null values in a sequence

```
(null, 1, null, 2)
```

Result (run with Zorba):

null 1 null 2

If an empty sequence is found as an object value, it is automatically converted to null.

Example 10.14. Automatic conversion to null.

```
{ "foo" : () }
```

Result (run with Zorba):

{ "foo" : null }

In an arithmetic opration or a comparison, if an operand is an empty sequence, an empty sequence is returned. If an operand is a null, an error is raised except for equality and inequality.

Example 10.15. Empty sequence in an arithmetic operation.

```
() + 2
```

Result (run with Zorba):

Example 10.16. Null in an arithmetic operation.

```
null + 2
```

Result (run with Zorba):

An error was raised: arithmetic operation not defined between types "js:null" and "xs:integer"

Example 10.17. Null and empty sequence in an arithmetic operation.

```
null + ()
```

Result (run with Zorba):

Example 10.18. Empty sequence in a comparison.

```
() eq 2
```

Result (run with Zorba):

Example 10.19. Null in a comparison.

```
null eq 2
```

Result (run with Zorba):

false

Example 10.20. Null in a comparison.

```
null lt 2
```

Result (run with Zorba):

true

Example 10.21. Null and the empty sequence in a comparison.

```
null eq ()
```

Result (run with Zorba):

Example 10.22. Null and the empty sequence in a comparison.

```
null lt ()
```

Result (run with Zorba):

Open Issues

The *JSON update syntax*[1] was not integrated yet into the core language. This is planned, and the syntax will be simplified (no json keyword, dot lookup allowed here as well).

The semantics for the *JSON serialization method*[2] is the same as in the JSONiq Extension to XQuery. It is still under discussion how to escape special characters with the Text output method.

[1] http://jsoniq.org/docs/JSONiqExtensionToXQuery/html/section-json-updates.html
[2] http://jsoniq.org/docs/JSONiqExtensionToXQuery/html/section-json-serialization.html

Appendix A. Revision History

Revision 1.0.11 **Tuesday Aug 13th, 2013** **Ghislain Fourny** ***g@28.io***

Refactored definition of parse-json to make options clear.
Fixed signature of size() to return "integer?".
Fixed signature of descendant-arrays() to return "array*".

Revision 1.0.10 **Thursday Aug 8th, 2013** **Ghislain Fourny** ***g@28.io***

Fixed semantics of libjn:project() to make it consistent with the other functions.
Added libjn:remove-keys() which does the contrary of libjn:project(), i.e., removes the given keys from all input objects.

Revision 1.0.9 **Monday July 22th, 2013** **Ghislain Fourny** ***g@28.io***

Relaxed the functions descendant-objects, descendant-pairs, values, flatten, project to accept any sequence of items. The behavior on mixed sequences is consistent with object and array lookup.
Also introduced new function descendant-arrays and relaxed size to accept the empty sequence.
Relaxed boolean value of sequences beginning with an object or array to be true.

Revision 1.0.8 **Tuesday June 18th, 2013** **Ghislain Fourny** ***g@28.io***

New array navigation syntax to avoid overloading dynamic function call syntax as well as predicate syntax: $array[[$position]] (was: $array($position)).
The functions keys() and members() were relaxed to take any sequence of items as parameters, performing implicit iteration, ignoring non-objects (keys) or non-arrays (members) (was: keys(object), members(array). Now: keys(item*), members(item*)).
New array unboxing syntax $array[], a shortcut for members($array). It also performs an implicit iteration.
Array lookup casts the position expression to an integer. Object lookup casts the key expression to a string (this reverts the change from 1.0.7)

Revision 1.0.7 **Monday May 27th, 2013** **Ghislain Fourny** ***g@28.io***

Relaxed array and object navigation to make it more XPath-like. No more cast is done. Lookup on mixed sequences of items (with an implicit iteration) are allowed. The empty sequence is returned in case of mismatch, i.e. in case of (i) integer lookup on object/atomic or (ii) string lookup on array/atomic.

Revision 1.0.6 **Friday May 24th, 2013** **Ghislain Fourny** ***g@28.io***

Reran all queries through a newer Zorba version (trunk 11475).

Revision 1.0.5 **Thursday May 2nd, 2013** **Ghislain Fourny** ***g@28.io***

Reran all queries through a newer Zorba version (trunk revision 11426).
Corrected some typos, like superfluous prefixes in function library sample queries.

Revision 1.0.4 Monday April 8th, 2013 Ghislain Fourny *g@28.io*

Allowed the context item to appear unparenthesized in an object lookup expression.
Added the function parse-json()

Revision 1.0.3 Thursday April 4th, 2013 Ghislain Fourny *g@28.io*

Fixed some typos.

Revision 1.0.2 Thursday April 4th, 2013 Ghislain Fourny *g@28.io*

Null are comparable with other atomic items and are considered the smallest.
Added link to semantics for all builtin functions inherited from XQuery.
Aggregated some examples.

Revision 1.0.1 Tuesday April 2nd, 2013 Ghislain Fourny *g@28.io*

Initial commit.

Index