JSONiq Extension to XQuery 1.0

Specification

XML and JSON: A Language To Rule Them All

Jonathan Robie

Ghislain Fourny

Matthias Brantner

Daniela Florescu

Till Westmann

Markos Zaharioudakis

JSONiq Extension to XQuery 1.0 Specification
XML and JSON: A Language To Rule Them All
Edition 1.0.3

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	Jonathan Robie	*jonathan.robie@gmail.com*
Editor	Ghislain Fourny	*ghislain.fourny@28msec.com*

The JSONiq extension to XQuery allows processing XML and JSON natively and with a single language. This extension is based on the same data model as the core JSONiq and is based on the same logical concepts. Because of the complexity of the XQuery grammar, the JSONiq extension to XQuery has a less pleasant syntax that the JSONiq core.

Status

The JSONiq extension to XQuery allows processing XML and JSON natively and with a single language. This extension is based on the same data model as the core JSONiq and is based on the same logical concepts. Because of the complexity of the XQuery grammar, the JSONiq extension to XQuery has a less pleasant syntax that the JSONiq core:

- Object lookup is done with the same syntax as dynamic function calls (because the dot is allowed in variable names and XPath name tests).
- All keys must be quoted (because unquoted strings are considered XPath name tests)
- Escaping in strings is done the XML way, i.e., with ampersands instead of backslashes.
- Builtin atomic types are actually builtin XML Schema types and must be prefixed with xs:
- The item type syntax for structured items like object(), array(), json-item(), item() must use parentheses, as item types without parentheses are considered user-defined atomic types in XQuery.
- The sequence type syntax for the empty sequence is empty-sequence().

Introduction

JSON and XML are both widely used for data interchange on the Internet. In many applications, JSON is replacing XML in Web Service APIs and data feeds; other applications support both formats. XML adds significant overhead for namespaces, whitespace handling, the oddities of XML Schema, and other things that are simply not needed in many data-oriented applications that require no more than simple serialization of program structures. On the other hand, many applications do need these features, and the ability to use document data together with traditional program data is important.

JSONiq is a small and simple set of extensions to XQuery that add support for JSON. For applications that need only JSON, we have defined a profile called XQ-- that removes all support for XML constructors, path expressions, user defined functions, and some other features considered unnecessary for most JSON queries. Syntax diagrams for XQ-- are available at *http://jsoniq.org/grammars/xq--/ui.xhtml*.

For applications that need to process XML together with JSON, we have defined a profile called XQ++ that adds these extensions to the full XQuery language. Syntax diagrams for XQ++ are available at *http://jsoniq.org/grammars/xq++/ui.xhtml*.

JSONiq consists of the following extensions to XQuery:

- Extensions to the XQuery Data Model (XDM) to support JSON.
- Support for JSON's datatypes, with a mapping to equivalent XML Schema types.
- Navigation for JSON Objects and JSON Arrays.
- Constructors for JSON Objects, Pairs, and JSON Arrays, using the same syntax as JSON.
- Update primitives against JSON items as well as updating expressions which produce them.
- New item types, which extend sequence type matching to allow the type of JSONiq datatypes to be specified in function parameters, return types, and other expressions that specify XQuery types.

JSONiq is designed to seamlessly integrate in XQuery's full composability paradigm. In particular, JSON constructors can appear in any XQuery expression, and any XQuery expression can appear in a JSON constructor.

The namespace **`http://jsoniq.org/functions`** is used for functions defined by this specification. This namespace is exposed to the user and is bound by default to the prefix **`jn`**. For instance, the function name **`jn:json-doc()`** is in this namespace.

The namespace **`http://jsoniq.org/types`** is used for types defined by this specification. This namespace is exposed to the user and is bound by default to the prefix **`jn`**. For instance, the type name **`js:null`** is in this namespace.

The namespace **`http://jsoniq.org/function-library`** is used for library functions defined by this specification. This namespace is exposed to the user, however no prefix is predeclared for this namespace. For convenience, this document uses the prefix **`libjn:`** for names in this namespace. For instance, the function name **`libjn:accumulate`** is in this namespace.

The namespace **`http://jsoniq.org/errors`** is used for the names of errors defined by this specification. This namespace is exposed to the user, however no prefix is predeclared for this namespace. For convenience, this document uses the prefix **`jerr:`** for names in this namespace. For instance, the error name **`jerr:JQTY0001`** is in this namespace.

The namespace **`http://jsoniq.org/updates`** is used for the names of update primitives defined by this specification. This namespace is not exposed to the user. For convenience, this document us-

es the prefix **`jupd:`** for names in this namespace. For instance, the update primitive **`jupd:delete-from-object`** is in this namespace.

Accessors used in JSONiq Data Model use the **`jdm:`** prefix. These functions are not exposed to the user and are for explanatory purposes of the data model within this document only. The **`jdm:`** prefix is not associated with a namespace.

JSONiq Data Model

JSONiq is based on the XQuery Data Model, adding support for JSON nulls, objects and arrays. [1]

To support nulls, JSONiq adds the following atomic datatype:

- *js:null*, which represents a JSON null.

To support JSON objects and arrays, JSONiq adds the following new items:

- *Object*, which represents a JSON object. An Object contains a set of string/item pairs.
- *Array*, which represent a JSON array. An Array contains a sequence of items.

As well as the generic term (associated with an item type):

- *JSON Item*, which can be an Object or an Array.

3.1. Simple Datatypes

JSON's simple values can have any of the following datatypes, which are supported as primitives in JSONiq.

- string
- number
- boolean (true, false)
- null

JSONiq represents a JSON string (a sequence of Unicode characters) as an xs:string. However, JSON supports strings (after resolving escaped characters) with characters of arbitrary unicode code-points, whereas XML 1.0 and XML 1.1 only support a subset of these. It is implementation-defined if unicode characters outside of XML 1.0 or XML 1.1 are supported. Implementations which support XML 1.1 (which only excludes the null character) are encouraged to add support for the null character to provide full-fledged JSON support.

JSONiq represents the JSON simple values 'true' and 'false' as **`xs:boolean`** values.

JSONiq adds one new data type: **`js:null`**, derived from **`xs:anyAtomicType`**. **`js:null`** has a singleton value space containing the value *null* [2]. The lexical representation of an instance of **`js:null`** is the string "**`null`**".

JSON defines number as follows:

> A number can be represented as integer, real, or floating point. JSON does not support octal or hex because it is minimal. It does not have values for NaN or Infinity because it does not want to be tied to any particular internal representation.

[1] In the XQ-- subset of JSONiq, we will define a simpler data model that omits much of the XQuery Data Model.
[2] (Because the typed value of all **`js:null`** instances is the same, nulls are always equal to each other for the purpose of value comparisons).

JSONiq uses XQuery's lexical representation of numbers to distinguish integer, real, and floating point numbers [3]:

```
NumericLiteral ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
```

If a numeric literal has no "." character and no e or E character, it has the atomic type **`xs:integer`**; if it has a "." character but no e or E character it has type **`xs:decimal`**; if it has an e or E character it has type **`xs:double`**.

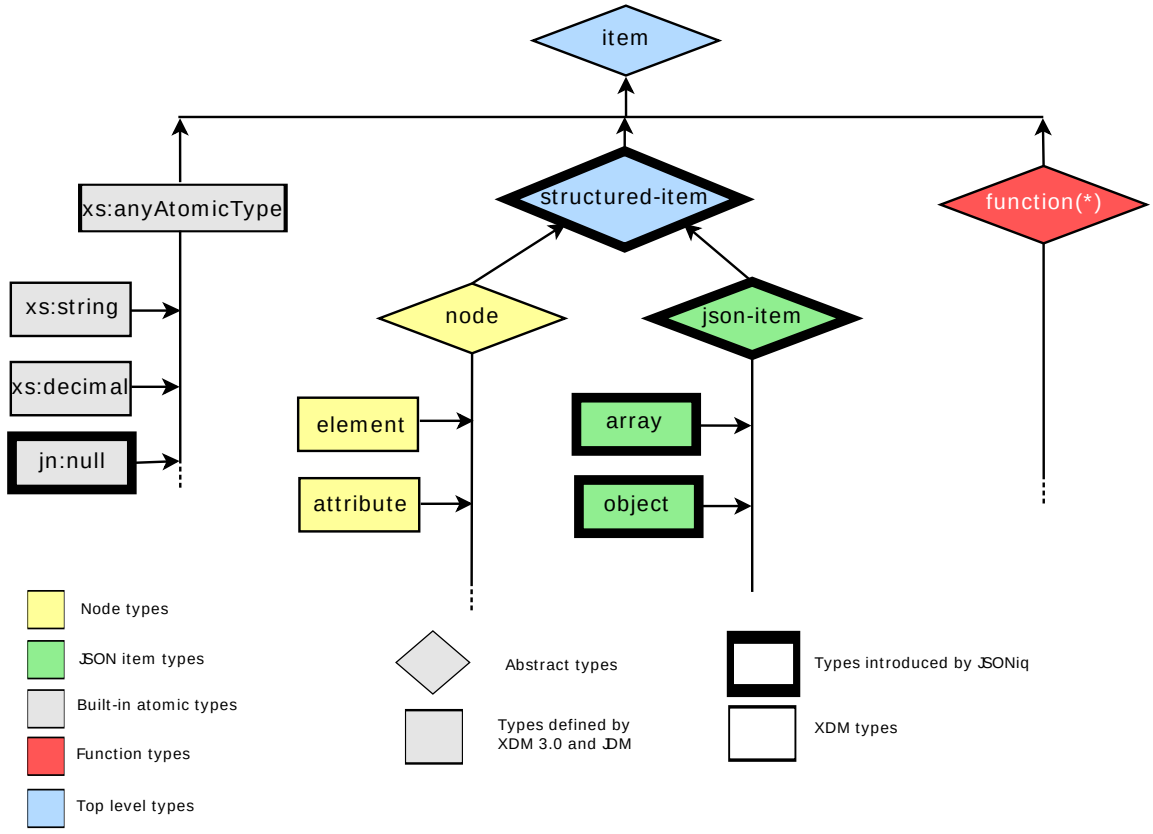
Number literals are mapped to atomic values according to the XML Schema specification.

3.2. JSON Items

A JSON Item is an item. A JSON Item has an identity [4] , and it can be serialized. Objects and Arrays are JSON Items [5] .

The diagram below shows the XQuery 3.0 type hierarchy, adding JSON Item and all types derived from it:

[3] The result of a JSONiq query may contain values such as NaN or Infinity, which cannot be serialized as JSON. These values have type **`xs:double`**, and are either rejected or treated in a special way during the JSON Serialization process. ECMAScript also has has Infinity and NaN, and takes a similar approach.
[4] The identity of an item is used only for updates, where it is used in the update primitives.
[5] In this document, the words *Object* and *Array* refer to JSON constructs. Unlike objects in most programming languages, a JSON object contains only data.



3.3. Objects

An Object represents a JSON object (a collection of string/value pairs).

Objects have the following property:

- *pairs*. A set of pairs. Each pair consists of an atomic value of type **`xs:string`** and of an item.

  [Consistency constraint: no two pairs have the same name (using **`fn:codepoint-equal`**).]

The XQuery data model uses accessors to explain the data model. Accessors are not exposed to the user and are only used for convenience in this specification. Objects have the following accessors:

- **`jdm:keys($o as object()) as xs:string*`**.

  Returns all keys in the object $o.

- **`jdm:value($o as object(), $s as xs:string) as item()`**.

Returns the value associated with $s in the object $o.

An Object does not have a typed value.

3.4. Arrays

An Array represents a JSON array (an ordered list of values).

Arrays have the following property:

- *members*. An ordered list of items.

Arrays have the following accessors:

- **`jdm:size($a as array()) as xs:integer`**

  Returns the number of values in the array $a.

- **`jdm:value($a as array(), $i as xs:integer) as item()`**

  Returns the value at position $i in the array $a.

An Array does not have a typed value.

3.5. ItemTypes for JSONiq Items

In XQuery, an ItemType is used to match certain items, for example in function signatures, return types, and a variety of other expressions. JSONiq extends XQuery's ItemType as follows:

```
ItemType ::= -- everything so far --
  | JSONTest
  | StructuredItemTest

JSONTest ::=
    JSONItemTest
  | JSONObjectTest
  | JSONArrayTest

StructuredItemTest ::= "structured-item" "(" ")"
JSONItemTest ::= "json-item" "(" ")"
JSONObjectTest ::= "object" "(" ")"
JSONArrayTest ::= "array" "(" ")"
```

The semantics of these expressions are straightforward.

- **`structured-item()`** matches any node or JSON Item.
- **`json-item()`** matches any JSON Item (Object or Array).
- **`object()`** matches any Object.
- **`array()`** matches any Array.

> Example 3.1. JSON Items in Function Signatures and instance-of expressions
> The following function returns any Objects that are members of an Array (see below for more on array navigation).

```
declare function local:objects-in-array($a as array()) as object()* {
  for $i in 1 to jn:size($a)
  let $item := $a($i)
  where $iteminstance of object()
  return $item
};
```

Construction of JSON values

JSONiq Constructors create Objects, and Arrays. If an Object Constructor or an Array Constructor contains only literal data, as a rule of thumb, the syntax is the same as the JSON serialization. Constructors can also contain XQuery expressions to create content as the result of a query.

Here is the syntax for object and array constructors:

```
PrimaryExpr ::= -- everything so far --
  | JSONConstructor

JSONConstructor ::=
    ArrayConstructor
  | ObjectConstructor

ArrayConstructor ::= "[" Expr? "]"

ObjectConstructor ::=
    "{" ( PairConstructor ("," PairConstructor)* )? "}"
  | | "{|" Expr "|}"
PairConstructor ::=  ExprSingle (":" | "?:") ExprSingle
```

XQuery expressions can occur within JSON constructors, and JSON constructors can occur within XQuery expressions. The semantics of such expressions are discussed in *Chapter 9, Combining XML and JSON*.

4.1. Array Constructors

The expression in an Array Constructor, if present, evaluates to a sequence of zero or more items. The members of the constructed array are copies of these items, in the same order.

Example 4.1. Array Constructors

An Array Constructor:

```
[ "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday" ]
```

Arrays can nest.

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

Combining an Array Constructor with XQuery expressions:

Query:

```
```

```
[ 10 to 15 ]
```

Result:

```
[ 10, 11, 12, 13, 14, 15 ]
```

4.2. Object Constructors

An Object Constructor is made of Pair Constructors. Each Pair Constructor creates a single string/item pair as follows:

- The string is the result of evaluating the left operand, atomizing it, checking that it results in a single atomic item, and casting it to **`xs:string`**. Errors such as **`jerr:JNTY0004`** or **`err:XPTY0004`** may be raised upon failure of one of these steps.
- If the right operand evaluates to a single item, the value of the pair is a copy of this result.

  If the right operand evaluates to the empty sequence, the value of the pair is the atomic value null (of type **`js:null`**). However, if the colon is preceded with a question mark, the pair is omitted instead.

  If the right operand evaluates to a sequence of more than one item, the value of the pair is an array containing copies of all items in this sequence, in the same order.

An object is constructed, the pairs property of which comprise all pairs generated by the pair constructors. An error **`jerr:JNDY0003`** is raised if two pairs have the same name (the comparison is made using **`fn:codepoint-equal`**).

Example 4.2. Object Constructors

An Object Constructor with literal data:

```
{
  "id" : 404,
  "name" : "Stanco Grease Pot",
  "price" : 6.49,
  "weight" : 3.8,
  "uses" : ["Grease storage","Backpacking pot"]
}
```

Combining an Object Constructor with XQuery expressions:

Query:

```
{
  "Sunday" : 1,
  "Monday" : 1 + 1,
  "Tuesday" : 3 * 1,
  "Wednesday" : 8 div 2,
  "Thursday" : 5,
  "Friday" : count(for $i in 1 to 6 return $i),
  "Saturday" : 10 - 3,
  "NotADay" ?: ()
}
```

Result:

```
{
  "Sunday" : 1,
  "Monday" : 2,
  "Tuesday" : 3,
  "Wednesday" : 4,
  "Thursday" : 5,
  "Friday" : 6,
  "Saturday" : 7
}
```

Note:

A JSON item cannot be a child of an XML element or attribute. If a JSON item is used in the content expression of an XQuery constructor, the result will raise an error, as described in *Chapter 9, Combining XML and JSON*.

There is also a syntax for dynamic object construction, which merges all the objects returned by the inner expression into a single object with a so-called "simple object union". A simple object union creates a new object, the pairs property of which is obtained by accumulating the pairs of all operand objects. An error **`jerr:JNDY0003`** is raised if two pairs with the same name are encountered.

Example 4.3. Dynamically building an object

Query:

```
let $object1 := { "Captain" : "Kirk" }
let $object2 := { "First officer" : "Spock" }
return {| $object1, $object2 |}
```

Result:

```
{
  "Captain" : "Kirk",
  "First officer" : "Spock"
}
```

Query:

```
{|
  for $d at $i in ("Sunday",
                   "Monday",
                   "Tuesday",
                   "Wednesday",
                   "Thursday",
                   "Friday",
                   "Saturday" )
  return { $d : $i }
|}
```

Result:

```
{
  "Sunday" : 1,
  "Monday" : 2,
```

```
            "Tuesday" : 3,
            "Wednesday" : 4,
            "Thursday" : 5,
            "Friday" : 6,
            "Saturday" : 7
          }
```

4.3. Strings

Strings can be constructed using XQuery String Literals.

Example 4.4. JSON String construction

```
          "This is a string"
```

4.4. Numbers

Numbers can be constructed using XQuery Integer, Decimal and Double literals.

Example 4.5. JSON Number construction

```
          42
          3.14
          6.022E23
```

4.5. Booleans

Booleans can be constructed using the XQuery builtin functions fn:true() and fn:false().

4.6. Null

Null can be constructed using the new builtin function jn:null().

4.7. Boolean and null literals

A new option **`jn:jsoniq-boolean-and-null-literals`** is available ("yes" or "no"), that activates or deactivates the interpretation of a standalone NameTest "true", "false" or "null" (meaning: occurring as a PathExpr, i.e., no slashes, no axes) as boolean and null literals. By default, it is set to "yes". [1]

[1] If this feature is activated, a relative path expression that tests for an element named **`true`**, **`false`**, or **`null`** must use the **`./`** construct, as in **`./true`**.

Navigation in JSON content

It is possible to navigate through JSON items using the dynamic function call syntax. The dynamic function call syntax is extended to handle object and array selection as follows:

- An implicit iteration is performed on the left-hand-side sequence. **`$sequence($params)`** is equivalent to:

```
for $item in $sequence return $item($params)
```

- The semantics of the dynamic function call is then defined on a single item, depending on its type:
  - If it is a function, the semantics is the same as defined in XQuery 3.0.
  - If it is an object, the dynamic function call is treated as an object lookup if unary, as a key listing if 0-ary. An error **`jerr:JNTY0018`** is raised if there is more than one parameter.
  - If it is an array, the dynamic function call is treated as an array lookup if unary, as an array unboxing if 0-ary. An error **`jerr:JNTY0018`** is raised if there is more than one parameter.
  - If it is an atomic or an XML node, the dynamic function call if unary or 0-ary always returns the empty sequence. An error **`jerr:JNTY0018`** is raised if there is more than one parameter.

5.1. Object lookup

If the dynamic function call is unary and the left-hand-side is an object, the semantics applied is that of object lookup and is as follows.

The unique parameter is cast to a string $s (an error is raised according to the semantics of casting if this fails).

If $s is in **`jdm:keys($o)`** then **`$o($s)`** returns the value of the pair with the name **`$s`**, i.e. **`jdm:value($o, $s)`**. Otherwise (i.e., it has no key matching **`$s`**), an empty sequence is returned.

Example 5.1. Object lookup

Retrieving a Pair by its name:

Query:

```
let $map := { "eyes" : "blue", "hair" : "fuchsia" }
return $map("eyes")
```

Result:

```
blue
```

Using Pairs from existing Objects to create a new Object:

Query:

```
let $x := { "eyes" : "blue", "hair" : "fuchsia" }
let $y := { "eyes" : brown, "hair" : "brown" }
```

```
        return { "eyes" : $x("eyes"), "hair" : $y("hair") }
```

Result:

```
          { "eyes" : "blue", "hair" : "brown" }
```

5.2. Object key listing

If the dynamic function call is 0-ary and the left-hand-side is an object, the semantics applied is that of key listing and is as follows.

`jdm:keys($o)` is returned.

Example 5.2. Object key listing

Retrieving the keys of an object:

Query:

```
       let $map := { "eyes" : "blue", "hair" : "fuchsia" }
       return $map()
```

Result:

```
       eyes hair
```

5.3. Array lookup

If the dynamic function call is unary and the left-hand-side is an array, the semantics applied is that of array lookup and is as follows.

The unique parameter is cast to an integer $i (an error is raised according to the semantics of casting if this fails).

If $i is comprised between 1 and **`jdm:size($a)`**, then **`$a($i)`** returns the value of the pair at position **`$i`**, i.e., **`jdm:value($a, $i)`**.

Otherwise (i.e., if **`$i`** is an invalid position), an empty sequence is returned.

Example 5.3. Array lookup

Query:

```
       let $wd := ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"]
       return $wd(1)
```

Result:

```
       Sunday
```

For the following queries, assume the variable **`$f`** is bound to the following two dimensional array.

Data:

```
[
  [ "mercury", "venus", "earth", "mars" ],
  [ "monday", "tuesday", "wednesday", "thursday" ]
]
```

Query:

```
$f(1)
```

Result:

```
[ "mercury", "venus", "earth", "mars" ]
```

Query:

```
$f(2)(2)
```

Result:

```
tuesday
```

5.4. Array unboxing

If the dynamic function call is 0-ary and the left-hand-side is an array, the semantics applied is that of array unboxing and is as follows.

`for $i in 1 to jdm:size($a) return $a($i)` is returned.

Example 5.4. Array unboxing

Query:

```
let $wd := ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"]
return $wd()
```

Result:

```
Sunday Mondy Tuesday Wednesday Thursday Friday Saturday
```

For the following queries, assume the variable **`$f`** is bound to the following two dimensional array.

Data:

```
        [
          [ "mercury", "venus", "earth", "mars" ],
          [ "monday", "tuesday", "wednesday", "thursday" ]
        ]
```

Query:

```
        $f()
```

Result:

```
        [ "mercury", "venus", "earth", "mars" ] [ "monday", "tuesday", "wednesday",
"thursday" ]
```

Builtin functions and operators

JSONiq defines a couple of basic builtin functions. Two of them define the atomized value and the effective boolean value of JSON items, and the others are for JSON navigation.

6.1. fn:boolean (aka Effective Boolean Value)

In XQuery, sequences can be converted to a boolean value. The effective boolean value is defined by invoking **`fn:boolean`**.

`fn:boolean($arg as item()*) as xs:boolean`

The definition of **`fn:boolean($arg)`** is changed as follows:

- If $arg is a sequence whose first item is an object or an array, **`fn:boolean`** returns true.
- If $arg is a singleton value of type **`js:null`**, **`fn:boolean`** returns false.

Example 6.1. Effective Boolean Value of JSONiq items

```
Examples
EBV( jn:null() )              false
EBV( { } )                    true
EBV ( { "foo": false } )      true
EBV ( { "foo": 3, "bar":4 } ) true
EBV( { "foo": 3 } )           true
EBV ( [1] )                   true
EBV ( ( [1], jn:null() ) )    true
```

6.2. fn:collection

This is a standard function in XQuery. JSONiq extends the function to also allow collections that contains JSON items, in an implementation-dependent order

`fn:collection($uri as xs:string?) as structured-item()*`

6.3. fn:data (aka Atomization)

In XQuery, sequences can be atomized. Atomization is defined by invoking fn:data.

`fn:data($arg as item()*) as xs:anyAtomicType*`

The error conditions of **`fn:data($arg)`** are extended as follows. An error is raised (**`jerr:JNTY0004`**) if an item in the sequence is an object or an array.

Example 6.2. Atomization of JSONiq items

```
fn:data( {"foo" : 3} )            jerr:JNTY0004
fn:data( [1] )                   jerr:JNTY0004
fn:data( {"foo" : 3, "bar" : 4 } ) jerr:JNTY0004
fn:data( { } )                   jerr:JNTY0004
```

6.4. fn:string (aka string value)

In XQuery, items can have string values, which can be obtained by invoking fn:string. Objects and arrays do not have a string value.

`fn:string($arg as item()?) as xs:string`

The error conditions of **`fn:string($arg)`** are extended as follows. An error is raised (**`jerr:JNTY0024`**) if the item is an object or an array.

Example 6.3. Atomization of JSONiq items

```
fn:string( {"foo" : 3} )           jerr:JNTY0024
fn:string( [1] )                  jerr:JNTY0024
fn:string( {"foo" : 3, "bar" : 4 } ) jerr:JNTY0024
fn:string( { } )                  jerr:JNTY0024
```

6.5. fn:trace

The default serialization method is changed to JSON-XML-hybrid.

6.6. jn:decode-from-roundtrip

This function decodes non-JSON types previously encoded with **`jn:encode-for-roundtrip.`**

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

`jn:decode-from-roundtrip($items as item()*, $options as object()) as item()*`

There is one optional parameter than can be supplied using the $options object.

- *prefix* (string): The prefix of the strings used to for the encoding (default: "Q{http://jsoniq.org/roundtrip}").

The above optional parameter is refered to as $prefix for convenience.

Error **`err:XPTY0004`** is raised in case one this parameter is of incorrect type.

Non-JSON values are encoded to a special object containing two required fields "[$prefix]type" (xs:string) and "[$prefix]value" (xs:string), and two optional fields "[$prefix]prefix" (xs:string) and "[$prefix]encoding-level (xs:integer, strictly positive). An object that has the two required fields, and that has no other field that these four fields, and whose values match the types, is called an Encoded Object.

This function maps each item of the input sequence to a new item in the output sequence, preserving order.

For convenience, the map is defined recursively from item() to item().

An array is mapped to an array the members of which are mapped resursively.

Atomic values and XML nodes are mapped to themselves.

An object is mapped depending on whether it is an Encoded Object (according to the above definition).

An Encoded Object whose "[$prefix]type" field is "document-node()", "element()", "text()", "comment()" or "processing-instruction()" is mapped to the XML node optained by parsing the field "[$prefix]value". An error **`jerr:JNTY0023`** is raised if the obtained node does not match the specified type.

An Encoded Object whose "[$prefix]type" field is another string is mapped to an atomic item with the type specified by the value of the field "[$prefix]type" (read as an EQName, with xs: automatically recognized as the XSD namespace).

- If this type is not xs:QName, xs:NOTATION or a subtype thereof, the atomic value is by casting the string in the field "[$prefix]value" to this type. An error **`jerr:JNTY0023`** if this case does not succeed.
- If this type is xs:QName, xs:NOTATION or a subtype thereof, then the atomic value to be built is a qualified name. The namespace and local name are built from the value (an EQName) of the field "[$prefix]value", and the prefix is built from the field "[$prefix]prefix" (empty if absent). An error **`jerr:JNTY0023`** if this does not succeed (value is not an EQName, prefix is not an NCName).

An object that is not an Encoded Object is mapped to an object with the same keys, and the associated values are mapped recursively.

Example 6.4. Encoding non-native JSON for roundtripping.

Query:

```
        jn:decode-from-roundtrip(
          {
            "nan" : {
              "Q{http://jsoniq.org/roundtrip}type" : "xs:double",
              "Q{http://jsoniq.org/roundtrip}value" : "NaN"
            },
            "inf" : {
              "Q{http://jsoniq.org/roundtrip}type" : "xs:double",
              "Q{http://jsoniq.org/roundtrip}value" : "INF"
            }
            "date" : {
              "Q{http://jsoniq.org/roundtrip}type" : "xs:date",
              "Q{http://jsoniq.org/roundtrip}value" : "1066-10-14"
             }
            "hat" : {
              "Q{http://jsoniq.org/roundtrip}type" : "Q{http://www.example.com/hat-
shop}hatSize",
              "Q{http://jsoniq.org/roundtrip}value" : "M"
            }
          }
        )
```

Result:

```
        {
          "nan" : xs:double("NaN"),
          "inf" : xs:double("INF"),
          "date" : xs:date("1066-10-14"),
          "user" : hat:hatSize("M")
        }
```

6.7. jn:encode-for-roundtrip

This function recursively encodes non-JSON types in such a way that they can be serialized as JSON while keeping roundtrip capability.

The following computations are made with respect to the static context of the caller, so that the schema type definitions are available.

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

`jn:encode-for-roundtrip($items as item()*, $options as object()?) as item()*`

There are two optional parameters than can be supplied using the $options object.

- *prefix* (string): The prefix of the strings used to for the encoding (default: "Q{http://jsoniq.org/roundtrip}").
- *serialization-parameters* (element node): The serialization parameters for any nested XML (as specified in the XQuery 3.0 specifications). By default:

```
            <serialization-parameters xmlns="http://www.w3.org/2010/xslt-xquery-serializa\
tion">
              <omit-xml-declaration value="yes" />
            </serialization-parameters>
```

The above optional parameters are refered to as $prefix and $serialization-parameters for convenience.

Error **`err:XPTY0004`** is raised in case one of the parameters is of incorrect type.

This function maps each item of the input sequence to a new item in the output sequence, preserving order.

For convenience, the map is defined recursively from item() to item().

Non-JSON values are encoded to a special object containing two required fields "[$prefix]type" (xs:string) and "[$prefix]value" (xs:string), and two optional fields "[$prefix]prefix" (xs:string) and "[$prefix]encoding-level (xs:integer, strictly positive). An object that has the two required fields, and that has no other field that these four fields, and whose values match the types, is called an Encoded Object.

An object that is not an Encoded Object is mapped to an object with the same keys, and the associated values are obtained by a recursive call on the original values.

An object that is an Encoded Object is left unchanged, except for the field "[$prefix]encoding-level" that is increased by 1 if it exists, or set to 2 if it is absent. This ensures round-trippability of already encoded data.

An array is mapped to an array. Its members are obtained with a recursive call on the members of the original array.

JSON native atomic values (of type exactly xs:double (except infinites and NaN), xs:decimal (except integers), xs:integer, xs:string, xs:boolean or js:null) are mapped to themselves.

XML nodes are mapped to an Encoded Object with two pairs:

- A pair named "[$prefix]type" and with the value "document-node()", "element()", "text()", "comment()" or "processing-instruction()", depending on the node kind. If it is an attribute or namespace node, **`err:SENR0001`** will be raised by the step below.
- A pair named "[$prefix]value" and whose value is a serialization of the XML node according to the XML output method and with the serialization parameters specified by $serialization-parameters.

Non-native atomic values that are not qualified names (i.e., neither xs:QName nor xs:NOTATION nor subtypes) are mapped to an Encoded Object with two fields:

- A field "[$prefix]type" containing the EQName of the atomic type (The XSD namespace may be simply output with the xs: prefix).
- A field "[$prefix]value" containing the lexical representation of the atomic value.

Non-native atomic values that are qualified names (i.e., xs:QName or xs:NOTATION or subtypes) are mapped to an object with two or three pairs:

- A field "[$prefix]type" containing the EQName of the atomic type (The XSD namespace may be simply output with the xs: prefix).
- A field "[$prefix]value" containing the EQName corresponding to the qualified name (which includes namespace and local name).
- A field "[$prefix]prefix" containing the prefix of the qualified name (only if there is such a prefix).

Example 6.5. Mapping non-JSON values

Query:

```
{
  "serialized XML" :
    jn:encode-for-roundtrip(<para>
         A pair named "[$prefix]value" (where [$prefix] is replaced with the
         value of the parameter $prefix) and whose value is a serialization
         of the XML node according to the XML output method and with the
         serialization parameters specified by $param.
         </para>)
}
```

Result:

```
{
  "serialized XML" : {
      "Q{http://jsoniq.org/roundtrip}type" : "node()",
      "Q{http://jsoniq.org/roundtrip}value" : "<para>
         A pair named "[$prefix]value" (where [$prefix] is replaced with the
         value of the parameter $prefix) and whose value is a serialization
         of the XML node according to the XML output method and with the
         serialization parameters specified by $param.
         </para>"
  }
}
```

Query:

```
        jn:encode-for-roundtrip({ "nan" : xs:double("NaN"),
                                  "inf" : xs:double("INF"),
                                  "date" : xs:date("1066-10-14"),
                                  "user" : hat:hatSize("M"),
                                  "QName" : xs:QName("jn:encode-for-roundtrip"),
                                  "EQName" : fn:QName("http://jsoniq.org/roundtrip", "val\
ue")
                                })
```

Result:

```
        {
          "nan" : {
            "Q{http://jsoniq.org/roundtrip}type" : "xs:double",
            "Q{http://jsoniq.org/roundtrip}value" : "NaN"
          },
          "inf" : {
            "Q{http://jsoniq.org/roundtrip}type" : "xs:double",
            "Q{http://jsoniq.org/roundtrip}value" : "INF"
          },
          "date" : {
            "Q{http://jsoniq.org/roundtrip}type" : "xs:date",
            "Q{http://jsoniq.org/roundtrip}value" : "1066-10-14"
          },
          "hat" : {
            "Q{http://jsoniq.org/roundtrip}type" : "Q{http://www.example.com/hat-
shop}hatSize",
            "Q{http://jsoniq.org/roundtrip}value" : "M"
          },
          "QName" : {
            "Q{http://jsoniq.org/roundtrip}type" : "xs:QName",
            "Q{http://jsoniq.org/roundtrip}value" : "Q{http://jsoniq.org/functions}encode-
for-roundtrip",
            "Q{http://jsoniq.org/roundtrip}prefix" : "jn"
          },
          "EQName" : {
            "Q{http://jsoniq.org/roundtrip}type" : "xs:QName",
            "Q{http://jsoniq.org/roundtrip}value" : "Q{http://jsoniq.org/roundtrip}value"
          }
        }
```

6.8. jn:json-doc

This function returns the JSON node associated with the supplied URI. It has the same semantics as **`fn:doc`**, except that it returns an Object or an Array.

```
jn:json-doc($uri as xs:string?) as json-item()?
```

6.9. jn:keys

This function returns all keys of all objects in the supplied sequence, using the jdm:keys accessor, with duplicates eliminated. The order in which the keys are returned is implementation-dependent but must be stable within a snapshot. Non-objects in the input sequence are ignored.

```
jn:keys($arg as item()*) as xs:string*
```

Example 6.6. Getting all keys in a sequence

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

Resulting (possibly with a different ordering) in:

```
          ("a", "b", "c")
```

Retrieving all Pairs from an Object:

Query:

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

Result:

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

6.10. jn:members

This functions returns all values that are in all arrays of an input sequence, preserving the order. Non-arrays in the input sequence are ignored.

```
       declare function jn:members($arg as item()*) as item()* {
         for $array in $arg[. instance of array()]
         for $position in 1 to jn:size($array)
         return $array($position)
       };
```

Example 6.7. Retrieving the members of an array

Query:

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

Result:

```
mercury venus earth mars 1 2 3
```

6.11. jn:null

This function returns the JSON null.

`jn:null() as js:null`

6.12. jn:parse-json

This function has the same semantics as fn:parse-xml(), except that it parses the string as JSON (not XML), and returns a sequence of objects or arrays rather than an XML document.

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

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

If **`$options("jsoniq-multiple-top-level-items")`** evaluates to the empty sequence, or the second parameter is not specified, it is considered to be true.

If **`$options("jsoniq-multiple-top-level-items")`** is not of type **`xs:boolean`**, then **`jerr:JNTY0020`** is raised..

If **`$options("jsoniq-multiple-top-level-items")`** is true, then jn:parse-json recognizes and parses sequences of objects and arrays, with no separation other than whitespaces.

If parsing is not successful, then **`jerr:JNDY0021`** is raised. Parsing is considered in particular to be non-successful if **`$options("jsoniq-multiple-top-level-items")`** is false and there is extra content after parsing a single abject or array.

Example 6.8. Parsing JSON items.

```
parse-json("{&amp;quot;foo&amp;quot; : &amp;quot;bar&amp;quot;} {&amp;quot;bar&amp;quot; : &amp;quot;foo&amp;quot;} [ 1,
 2, 3 ]")
```

Result:

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

6.13. jn:size

This function returns the size of the supplied array using the jdm:size accessor (or the empty sequence if the empty sequence is provided).

`jn:size(array()?) as xs:integer?`

Example 6.9. Retrieving the size of an array

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

Result:

```
10
```

6.14. Changes to cast semantics

The semantics of casts is changed as follows.

No value of an atomic type different than js:null (or derived) can be cast to js:null. An error **err:XPTY0004** is raised instead.

A js:null can still be cast to xs:string or xs:untypedAtomic as specified in the XPath function specification, which leads to its canonical representation "null".

6.15. Changes to arithmetic operation semantics

The semantics of arithmetic operations is changed as follows. If one of the operands is null, an type error **err:XPTY0004** is raised.

Example 6.10. Addition

Query

```
null + 1
```

Result

```
Error.
```

6.16. Changes to value comparison semantics

The semantics of value comparison is changed as follows.

Null is considered equal to itself, unequal to any other atomic items, and smaller than any other atomic item.

Example 6.11. Value comparison

Query

```
1 eq null, null eq null, 1 ne null, null lt 1
```

Result

```
false true true true
```

6.17. Changes to general comparison semantics

The semantics of general comparison is changed as follows.

After atomization, if one of the operands is the null atomic item, the other item is not cast, and pairwise comparison is done using the modified semantics of value comparison.

Example 6.12. General comparison: equality

Query

```
(null, 2) = (1, 3)
```

Result

```
false
```

JSON updates

JSONiq introduces new update primitives for updating Objects and Arrays. Update primitives can be generated with new JSONiq updating expressions.

An individual function may create an invalid JSON instance; however, an updating query must produce a valid JSON instance once the entire query is evaluated, or an error is raised and the entire update fails, leaving the instance in its original state.

7.1. JSON udpate primitives

The following new update primitives are introduced.

- **`jupd:insert-into-object($o as object(), $p as object())`**

  Inserts all pairs of the object **`$p`** into the object $o.

- **`jupd:insert-into-array($a as array(), $i as xs:integer, $c as item()*)`**

  Inserts all items in the sequence $c before position $i into the array $a.

- **`jupd:delete-from-object($o as object(), $s as xs:string*)`**

  Removes the pairs the names of which appear in $s from the object $o.

- **`jupd:delete-from-array($a as array(), $i as xs:integer)`**

  Removes the item at position $i from the array $a (causes all following items in the array to move one position to the left).

- **`jupd:replace-in-array($a as array(), $i as xs:integer, $v as item())`**

  Replaces the item at position $i in the array $a with the item $v (do nothing if $i is not comprised between 1 and jdm:size($a)).

- **`jupd:replace-in-object($o as object(), $n as xs:string, $v as item())`**

  Replaces the value of the pair named $n in the object $o with the item $v (do nothing if there is no such pair).

- **`jupd:rename-in-object($o as object(), $n as xs:string, $p as xs:string)`**

  Renames the pair originally named $n in the object $o as $p (do nothing if there is no such pair).

Update primitives within a PUL are applied with strict snapshot semantics. For examples, the positions are resolved against the array before the updates. Names are resolved on the object before the updates.

In the middle of a program, several PULs can be produced against the same snapshot. They are then merged with **`upd:mergeUpdates`**, which is extended as follows.

- Several deletes on the same object are replaced with a unique delete on that object, with a list of all selectors (names) to be deleted, where duplicates have been eliminated.
- Several deletes on the same array and selector (position) are replaced with a unique delete on that array and with that selector.

- Several inserts on the same array and selector (position) are equivalent to a unique insert on that array and selector with the content of those original inserts appended in an implementation-dependent order (like XQUF).
- Several inserts on the same object are equivalent to a unique insert where the objects containing the pairs to insert are merged. An error **`jerr:JNUP0005`** is raised if a collision occurs.
- Several replaces on the same object or array and with the same selector raise an error **`jerr:JNUP0009`**.
- Several renames on the same object and with the same selector raise an error **`jerr:JNUP0010`**.
- If there is a replace and a delete on the same object or array and with the same selector, the replace is omitted in the merged PUL.
- If there is a rename and a delete on the same object or array and with the same selector, the rename is omitted in the merged PUL.

At the end of an updating program, the resulting PUL is applied with **`upd:applyUpdates`**, which is extended as follows:

- First, before applying any update, each update primitive (except the jupd:insert-into-object primitives, which do not have any target) locks onto its target by resolving the selector on the object or array it updates. If the selector is resolved to the empty sequence, the update primitive is ignored in step 2. After this operation, each of these update primitives will contain a reference to either the pair (for an object) or the value (for an array) on or relatively to which it operates
- Then each update primitive is applied, using the target references that were resolved at step 1. The order in which they are applied is not relevant and does not affect the final instance of the data model. After applying all updates, an error **`jerr:JNUP0006`** is raised upon pair name collision within the same object.

7.2. Update syntax: new updating expressions

The following syntax is introduced for updates.

7.2.1. Deleting expressions

```
JSONDeleteExpr ::= "delete" "json" PrimaryExpr ( "(" ExprSingle ")" )+
```

PrimaryExpr followed by all "(" ExprSingle ")" except the last one, is evaluated according to the semantics of dynamic function calls. It must return a single object $o or a single array $a. Otherwise, **`jerr:JNUP0008`** is raised. After this evaluation, two cases can appear for interpreting the last "(" ExprSingle ")":

(for explanatory purposes, suboperands are replaced with a variable containing the result of their evaluation)

- **`delete json $o($s)`**

  $s is atomized and cast to xs:string (**`jerr:JNUP0007`** is raised upon failure).

  Creates the update primitive **`jupd:delete-from-object($o, $s)`**. An error **`JNUP:0016`** is raised if $o does not contain a pair with the key $s.

- **`delete json $a($i)`**

  $i is atomized and cast to xs:integer (**`jerr:JNUP0007`** is raised upon failure).

  Creates the update primitive **`jupd:delete-from-array($a, $i)`**. An error **`JNUP:0016`** is raised if $i is out of the range of the array $a.

> Example 7.1. Deleting from an object and from an array
>
> ```
> delete json $o("foo")
> delete json $a(2)
> ```

Note:

If **`$o($s)`** or **`$a($i)`** resolves to an empty sequence, the produced update primitive will have no effect.

7.2.2. Inserting expressions

```
JSONInsertExpr ::= "insert" "json" ExprSingle "into" ExprSingle ("at" "position"
ExprSingle)?
                       | "insert" "json" PairConstructor ("," Pair Constructor)* "into"
ExprSingle
```

(for explanatory purposes, suboperands are replaced with a variable containing the result of their evaluation)

The second variant corresponds to the insertion of a statically known number of pairs:

`insert json "foo" : "bar", "bar" : "foo" into $o`

is defined as being equivalent to:

`insert json { "foo" : "bar", "bar" : "foo" } into $o`

Inserting expressions are available in two flavors (object insertion, array insertion):

- **`insert json $p into $o`**

  $o must be an object. Otherwise, **`jerr:JNUP0008`** is raised.

  $p must be a sequence of objects. Otherwise, **`jerr:JNUP0019`** is raised.

  $o is post-processed by invoking **`jn:object`**, which results in a single object containing all pairs to insert. In particular, **`jerr:JNDY0003`** is raised upon a pair collision.

  Makes a copy $p and creates the update primitive **`jupd:insert-into-object($o, $p)`**

- **`insert json $c into $a at position $i`**

  $a must be an array. Otherwise, **`jerr:JNUP0008`** is raised.

  $i is atomized and cast to xs:integer (**`jerr:JNUP0007`** is raised upon failure).

  Makes a copy of all items in $c and creates the update primitive **`jupd:insert-into-array($a, $i, $c)`**

Example 7.2. Inserting into an array or an object

```
insert json (1, 2, 3) into $a at position 3
insert json { "foo": 3, "bar":4 } into $o
```

7.2.3. Renaming expressions

```
JSONRenameExpr ::= "rename" "json" PrimaryExpr  ( "(" ExprSingle ")" )+ "as" Ex\
prSingle
```

(for explanatory purposes, suboperands are replaced with a variable containing the result of their evaluation)

`rename json $o($s) as $n`

PrimaryExpr followed by all "(" ExprSingle ")" except the last one, is evaluated according to the semantics of dynamic function calls. It must return a single object $o. Otherwise, **`jerr:JNUP0008`** is raised. After this evaluation, the last "(" ExprSingle ")" is interpreted as follows:

$s is atomized and cast to xs:string (**`jerr:JNUP0007`** is raised upon failure).

Creates the update primitive **`jupd:rename-in-object($o, $s, $n)`**. An error **`JNUP:0016`** is raised if $o does not contain a pair with the key $s.

Example 7.3. Renaming an object pair

```
rename json $o("foo") as "bar"
```

7.2.4. Replacing expressions

```
JSONReplaceExpr ::= "replace" "json" "value" "of" PrimaryExpr ( "(" ExprSingle
")" )+ "with" ExprSingle
```

PrimaryExpr followed by all "(" ExprSingle ")" except the last one, is evaluated according to the semantics of dynamic function calls. It must return a single object $o or a single array $a. Otherwise, **`jerr:JNUP0008`** is raised. After this evaluation, two cases can appear for interpreting the last "(" ExprSingle ")":

(for explanatory purposes, suboperands are replaced with a variable containing the result of their evaluation)

- **`replace json value of $o($s) with $c`**

  $s is atomized and cast to xs:string (**`jerr:JNUP0007`** is raised upon failure).

  $c is postprocessed in the same way as values in a pair constructor, i.e., if $c is the empty sequence, it is replaced with the atomic value null, and if it is a sequence of more than one item, an array is created with all items in this sequence and in the same order. Ultimately, in all cases, $c will be a single item.

Makes a copy of $c and creates the update primitive **`jupd:replace-in-object($o, $s, $c)`**. An error **`JNUP:0016`** is raised if $o does not contain a pair with the key $s.

- **`replace json value of $a($i) with $c`**

  $i is atomized and cast to xs:integer (**`jerr:JNUP0007`** is raised upon failure).

  $c is postprocessed in the same way as values in a pair constructor, i.e., if $c is the empty sequence, it is replaced with the atomic value null, and if it is a sequence of more than one item, an array is created with all items in this sequence and in the same order. Ultimately, in all cases, $c will be a single item.

  Makes a copy of $c and creates the update primitive **`jupd:replace-in-array($a, $i, $c)`**. An error **`JNUP:0016`** is raised if $i is out of the range of array $a.

> Example 7.4. Replace a value in an object or in an array
>
> ```
> replace json value of $o("foo") with 5
> replace json value of $a(3) with 25
> ```

7.2.5. Appending expressions

```
JSONAppendExpr ::= "append" "json" ExprSingle "into" ExprSingle
```

(for explanatory purposes, suboperands are replaced with a variable containing the result of their evaluation)

`append json $c into $a`

$a must be an array. Otherwise, **`jerr:JNUP0008`** is raised.

Creates the update primitive **`insert-into-array($a, $a()+1, $c)`**

> Example 7.5. Appending values to an array
>
> ```
> append json (1,2) into $a
> ```

Function library

This section defines a function library on top of JSONiq. These functions are not part of the JSONiq core, because the latter is intended to be minimal.

8.1. libjn:accumulate

This function dynamically builds an object, like jn:object, except that it does not throw an error upon pair collision. Instead, it accumulates them into an array. Non-object items in the input sequence are ignored.

```
declare function libjn:accumulate($sequence as item()*) as object()
{
  {|
    for $key in $sequence() return { $key : $sequence($key) }
  |}
};
```

8.2. libjn:descendant-arrays

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

```
declare function libjn:descendant-arrays($sequence as item()*) as array()*
{
  for $item in $sequence
  return typeswitch ($item)
  case array() return (
    $item,
    libjn:descendant-arrays($item())
  )
  case object() return
    libjn:descendant-arrays(libjn:values($item))
  default return ()
};
```

Example 8.1. Accessing all descendant arrays

Query:

```
libjn:descendant-arrays(
  (
    { "foo" : { "bar" : [ 1, 2 ] } },
    [ [ { "foo" : "bar", "bar" : "foo" } ] ],
    true(),
    1,
    jn:null()
  )
)
```

Result:

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

8.3. libjn:descendant-objects

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

```
declare function libjn:descendant-objects($sequence as item()*) as object()*
{
  for $item in $sequence
  return typeswitch ($item)
  case object() return (
    $item,
    libjn:descendant-objects(libjn:values($item))
  )
  case array() return
    libjn:descendant-objects($item())
  default return ()
};
```

Example 8.2. Accessing all descendant objects

Query:

```
libjn:descendant-objects(
  (
    { "foo" : { "bar" : [ 1, 2 ] } },
    [ [ { "foo" : "bar", "bar" : "foo" } ] ],
    true(),
    1,
    jn:null()
  )
)
```

Result:

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

8.4. libjn:descendant-pairs

This function returns all descendant pairs within the supplied items.

```
      declare function libjn:descendant-pairs($sequence as item()*)
      {
        for $item in $sequence
        return typeswitch ($item)
        case object() return
          for $key in jn:keys($item)
          let $value := $item($key)
          return (
            { $key : $value },
            libjn:descendant-pairs($value)
          )
        case array() return
          libjn:descendant-pairs($item())
        default return ()
      };
```

Example 8.3. Accessing all descendant pairs

Query:

```
libjn:descendant-pairs(
  (
    { "foo" : { "bar" : [ 1, 2 ] } },
    [ [ { "foo" : "bar", "bar" : "foo" } ] ],
    true(),
    1,
    jn:null()
  )
)
```

Result:

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

Example 8.4. Query all pairs with a given name.

Query:

```
libjn:descendant-pairs({
  "first" : 1,
  "second" : {
    "first" : "a",
    "second" : "b"
  }
})("first")
```

Result:

```
       1 a
```

8.5. libjn:flatten

This function recursively "flattens" the supplied (top-level) arrays, leaving non-arrays unchanged.

```
declare function libjn:flatten($sequence as item()*) as item()*
{
  for $item in $sequence
  return
    typeswitch ($item)
    case array() return libjn:flatten($item())
    default return $item
};
```

Example 8.5. Flattening arrays.

Query:

```
libjn:flatten(
```

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

Result:

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

8.6. libjn:intersect

This function returns an object that only contains keys shared by all input objects, and associates to each of these keys the corresponding values in the input objects (wrapped in an array if more than one). Non-objects are ignored.

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

Example 8.6. Intersecting objects.

Query:

```
libjn:intersect(
  (
    { "foo" : { "bar" : [ 1, 2 ] } },
    [ 1, 2, 3, 4 ],
    { "foo" : "bar", "bar" : "foo" }
    true(),
    1,
    jn:null()
  )
)
```

Result:

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

8.7. libjn:project

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

```
declare function libjn:project($sequence as item()*, $keys as xs:string*) as item()*
{
  for $item in $sequence
  return typeswitch ($item)
         case $object as object() return
         {|
           for $key in $object()[ . = $keys ]
           return { $key : $object($key) }
         |}
         default return $item
};
```

Example 8.7. Projecting objects.

Query:

```
libjn:project(
  (
    {
      "Captain" : "Kirk",
      "First Officer" : "Spock",
      "Engineer" : "Scott"
    },
    [ 1, 2, 3, 4 ]
    {
      "Captain" : "Archer",
      "Engineer" : "Trip",
    },
    true(),
    1,
    jn:null()
  ),
  ("Captain", "First Officer", "XQuery Evangelist")
)
```

Result:

```
{ "Captain" : "Kirk", "First Officer" : "Spock" }
[ 1, 2, 3, 4 ] { "Captain" : "Archer" } true 1 null
```

8.8. libjn: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 libjn:remove-keys($sequence as item()*, $keys as xs:string*) as item()*
{
  for $item in $sequence
  return typeswitch ($item)
         case $object as object() return
         {|
           for $key in $object() [ not (. = $keys) ]
           return { $key : $object($key) }
         |}
         default return $item
};
```

Example 8.8. Projecting objects.

Query:

```
libjn:remove-keys(
  (
    {
      "Captain" : "Kirk",
      "First Officer" : "Spock",
      "Engineer" : "Scott"
    },
    [ 1, 2, 3, 4 ]
    {
      "Captain" : "Archer",
      "Engineer" : "Trip",
    },
    true(),
    1,
    jn:null()
  ),
  ("Captain", "First Officer", "XQuery Evangelist")
)
```

Result:

```
{ "Engineer" : "Scott" } [ 1, 2, 3, 4 ] { "Engineer" : "Trip" } true 1 null
```

8.9. libjn:values

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

```
      declare function libjn:values($sequence as item()*) as item()* {
        for $item in $sequence
        return $item() ! $item(.)
      };
```

Example 8.9. Extracting all values from objects.

Query:

```
libjn:values(
  (
    {
      "Captain" : "Kirk",
      "First Officer" : "Spock",
      "Engineer" : "Scott"
    },
    [ 1, 2, 3, 4 ]
    {
      "Captain" : "Archer",
      "Engineer" : "Trip",
    },
    true(),
    1,
    jn:null()
  )
)
```

Result:

```
Kirk Spock Scott Archer Trip
```

Combining XML and JSON

JSONiq is designed to allow XML and JSON to be used in the same query.

The syntax of JSONiq allows JSON constructors to contain XML values, and allows JSON constructors to occur in XML constructors. JSON does not support XML nodes or types, and XML does not support Objects or Arrays, but JSONiq allows Objects and Arrays to contain XML nodes, and defines rules for using JSONiq nodes in XML content expressions.

Example 9.1. XML in JSON

Both XML nodes and atomic values may occur in the values of Objects.

```
        {
          "element" : <mercury>Hg</mercury>
,
          "atomic value" : xs:date("1896-01-24")
          "several dates" : [ xs:date("1066-10-14"), xs:date("1935-01-11"),
 xs:date("1989-11-09") ]
        }
```

XML nodes and atomic values may also appear in Arrays.

```
[ xs:date("1066-10-14"), <mercury>Hg</mercury>
, "ice cream" ]
```

JSONiq does not allow XML nodes to contain Objects and Arrays. If an XQuery element content sequence, the value of the enclosed expression of an attribute, or the value of the content expression of a computed constructor contains an Object or Array, an error **`jerr:JNTY0011`** is raised.

Example 9.2. Objects in XML Constructors

Objects can be indirectly used in the content expression of any XQuery constructor.

Query:

```
        let $object := { "x" : 10, "y" : 12 }
        let $x := $object("x")
        return <x>{ $x }</x>
```

An Array can also be used in the content expression.[1]:

```
        <svg><polygon stroke="blue" points="(jnlib:flatten([ [10,10], [20,10], [20,20],
 [10,20] ])}" /></svg>
```

Here is the result of the above query:

```
<svg><polygon stroke="blue" points="10 10 20 10 20 20 10 20" /></svg>
```

[1] The data in this example is taken from an example on Stefan Goessner's JSONT site (*http://goessner.net/articles/jsont/*).

JSON Serialization

JSONiq defines two new output methods: "JSON" and "JSON-XML-hybrid". The JSON output method outputs guaranteed JSON (application/json) with the small exception that it may also output a sequence of objects and arrays (as seems to be standard practice, for example, in REST APIs). The JSON-XML-hybrid method allows serializing mixed sequences of JSON items and XML nodes for convenience. The JSON-XML-hybrid method is the default output method.

10.1. New serialization parameters

One new serialization parameter is introduced:

- **`jsoniq-multiple-top-level-items`** (one of: "yes", "no"):

  Specifies if multiple items are allowed. Default is "yes".

10.2. Changes to sequence normalization

The XQuery and XSLT serialization specification performs sequence normalization before serializing an XDM instance. The result is a sequence of exactly one document node.

JSONiq extends sequence normalization as follows:

The final step of building a document node with all items as its children is only done for adjacent XML nodes. JSON objects and arrays in the input sequence are left untouched.

This means that the final result of sequence normalization will be a sequence of objects, arrays and document nodes.

10.3. The JSON output method

10.3.1. Serialization of a sequence of items

If the normalized sequence does not consist exclusively of JSON arrays and objects, an error **`jerr:JNSE0014`** is raised.

If jsoniq-multiple-top-level-items is set to "no" and the sequence to be serialized does not consist of a single object or array, an error **`jerr:JNSE0012`** is raised.

If jsoniq-multiple-top-level-items is set to "yes", then the JSON items in the sequence are serialized one by one as specified below, in the same order and without any separating character (except white-spaces, depending on the indent parameter).

10.3.2. Serialization of individual JSON values

This section specifies how JSON items as well as JSON atomic values are serialized.

- Object

  An object is serialized as an opening curly brace {, followed by the comma-separated serializations of all its pairs, followed by a closing curly brace.

  A pair is serialized as the serialization of its name, followed by a colon, followed by the serialization of its value.

- Array

  An array is serialized as an opening square brace, followed by the comma-separated serializations of all its member values, followed by a closing square brace.

- **`xs:string`**

  An atomic value of type xs:string is serialized between double quotes and with the escaping specified in http://www.ietf.org/rfc/rfc4627.txt.

- **`xs:double`**, **`xs:float`** (except Nan, positive/negative infinity), **`xs:decimal`** or subtype

  An atomic value of a numeric type (**`xs:double`**, **`xs:float`**, **`xs:decimal`** or subtype) is serialized as its lexical representation, i.e., is cast to **`xs:string`**, not surrounded by quotes. This corresponds to the JSON number notation, but with a few exceptions which must be fixed before serializing:

  - The leading + sign must be removed if present.
  - A zero must be added in front of and/or after the dot if no digit is present in front of and/or after the dot.
  - Unnecessary zeros in front of the number (except if the only digit in front of the dot is 0) must be removed.
  - Values of type xs:decimal that have a lexical representation without dot must be suffixed with ".0" (so that it parses back as a decimal).
  - Values of type xs:double that have a lexical representation without E must be suffixed with "E0" (so that it parses back as a double).

- **`xs:boolean`** or subtype

  An atomic value matching the type **`xs:boolean`** is serialized as **`true`** or **`false`**, not surrounded by quotes.

- **`js:null`** or subtype

  The atomic value null matching the type **`js:null`** is serialized as **`null`**, not surrounded by quotes.

- All other atomic values are output as if they were of value **`xs:string`** (i.e., as a JSON string).

- An error jerr:JNSE0014 is raised if an XML node or a function is encountered.

10.3.3. Influence of other serialization parameters upon the JSON output method

10.3.3.1. JSON output method: the indent parameter

Like for the XML output method, the indent parameter controls if the serializer is allowed to add extra whitespaces to make the output easier to read. If the indent parameter has the value "yes", the serializer may output additional whitespaces characters outside of JSON literals. If the indent parameter has the value "no", the serializer may not output any whitespaces outside of JSON literals.

If the indent parameter is set to yes, the algorithm for adding whitespaces is allowed to add them at the following places, and only at these places:

- Between an object-opening curly brace and the first key.

- On either side of a key-value-separating colon.
- Before and after pair-separating commas in an object.
- Between the value of the last pair of an object and the object-closing curly brace.
- Between an array-opening square bracket and the first member.
- Before and after value-separating commas in an array.
- Between the last member of an array and the array-closing square bracket.
- Between two objects or arrays if multiple JSON items are serialized.

In particular, no whitespaces may be added inside JSON literals (strings, numbers, booleans, nulls).

If the indent parameter is set to no, no whitespace may be output to any of the places mentioned above.

10.3.3.2. JSON output method: the byte-order-mark parameter

The byte-order-mark parameter is applicable to the JSON output method and has the same semantics as for the XML output method.

10.3.3.3. JSON output method: the use-character-maps parameter

The use-character-maps parameter is applicable to the JSON output method and has the same semantics as for the XML output method.

Like for the XML output method, the result of serialization using the JSON output method is not guaranteed to be well-formed JSON if character maps have been specified.

10.4. The JSON-XML-hybrid output method

This method serializes a sequence of items to a stream of objects, arrays, XML nodes and text as follows:

- An object or array is serialized according to the JSON output method (the value of jsoniq-multiple-top-level-items is irrelevant), with the exception that no error is raised if an XML node is found as a value in an object or a member of an array. Instead, the XML node is serialized to a JSON string as described below for document nodes.
- A document node is serialized according to the xml output method and the corresponding serialization parameters (except that the parameters setting declaration and doctypes are ignored, and no doctype or XML declaration is output).
- The item serialization outputs are not separated by any spaces in the final output (except if the indent parameter is set, in which case the serializer is free to insert whitespaces).

10.5. Changes to ther other output methods

The other outputs methods (XML, HTML, XHTML, Text) are changed as follows: if the normalized sequence does not consist of exactly one document node, then **`jerr:JNSE0022`** is raised.

Error codes

The JSONiq error codes are summarized here.

jerr:JNDY0003

It is a dynamic error if two pairs in an object constructor or in a simple object union have the same name.

jerr:JNTY0004

Arrays and objects cannot be atomized. It is a type error to call fn:data on a sequence containing an array or an object.

jerr:JNUP0005

It is a dynamic error if a pending update list contains two JSON insert update primitives on the same object and pair name.

jerr:JNUP0006

It is a dynamic error if upd:applyUpdates causes an object to contain two pairs with the same name.

jerr:JNUP0007

It is a type error if, in an updating expression, an array selector cannot be cast to xs:integer or if an object selector cannot be cast to xs:string.

jerr:JNUP0008

It is a dynamic error if the target of a JSON delete or JSON replace expression is not an array or an object. It is a dynamic error if the target of a renaming expression is not an object. It is a dynamic error if the target of an appending expression is not an array. It is a dynamic error if the target of a position-inserting expression is not an array. It is a dynamic error if the target of a non-position-inserting expression is not an object.

jerr:JNUP0009

It is a dynamic error if a pending update list contains two JSON replacing update primitives on the same object or array, and with the same selector.

jerr:JNUP0010

It is a dynamic error if a pending update list contains two JSON renaming update primitives on the same object and with the same selector.

jerr:JNTY0011

It is a type error if the content sequence in a node constructor or in an XQUF insert or replace update expression contains an object or an array.

jerr:JNSE0012

It is a dynamic error to serialize something else than a unique JSON item with the JSON output method if the jsoniq-multiple-top-level-items is set to no.

jerr:JNSE0014

It is a dynamic error to serialize a function, a node or a standalone atomic item with the JSON output method.

jerr:JNSE0015

It is a dynamic error to serialize a top-level item which is not a JSON item if the jsoniq-multiple-top-level-items serialization parameter is not set to "array".

jerr:JNUP0016

It is a dynamic error if it is attempted to create a replace, delete or rename update primitive with a selector that cannot be resolved against the target array or object.

jerr:JNTY0018

It is a type error if there is not exactly one supplied parameter for an object or array selector.

jerr:JNUP0019

It is a dynamic error if the content expression, in an object insert expression, does not evaluate to a sequence of objects.

jerr:JNTY0020

It is a dynamic error if, when calling jn:parse-json, the option "jsoniq-multiple-top-level-items" is not a boolean.

jerr:JNDY0021

It is a dynamic error if parsing a string to one or several objects or arrays is not successful, or if it results in more than one item and "jsoniq-multiple-top-level-items" is false.

jerr:JNSE0022

It is a dynamic error if an object or array is encountered upon serializing as XML, XHTML, HTML or Text.

jerr:JNTY0023

It is a dynamic error if, when calling jn:decode-from-roundtrip, the type specified in an Encoded Object is not recognized, or if the value cannot be cast to it (if it is an atomic type) or if the parsed XML node does match it (if it is an XML node type).

jerr:JNTY0024

Arrays and objects do not have a string value. It is a type error to call fn:string on an array or an object.

Grammar Summary

```
PrimaryExpr ::=  -- everything so far --
  | ObjectConstructor
  | ArrayConstructor

ObjectConstructor ::=  "{" ( PairConstructor ("," PairConstructor)* )? "}"

PairConstructor ::=  ExprSingle ":" ExprSingle

ArrayConstructor ::=  "[" Expr? "]"

ExprSingle ::=  -- everything so far --
  | JSONDeleteExpr
  | JSONInsertExpr
  | JSONRenameExpr
  | JSONReplaceExpr
  | JSONAppendExpr

JSONDeleteExpr ::= "delete" "json" PrimaryExpr ( "(" ExprSingle ")" )+

JSONInsertExpr ::= "insert" "json" ExprSingle "into" ExprSingle
                   ("at" "position" ExprSingle)?
                 | "insert" "json" PairConstructor ("," Pair Constructor)*
                   "into" ExprSingle

JSONRenameExpr ::= "rename" "json" PrimaryExpr  ( "(" ExprSingle ")" )+
                   "as" ExprSingle

JSONReplaceExpr ::= "replace" "json" "value" "of"
                    PrimaryExpr ( "(" ExprSingle ")" )+
                    "with" ExprSingle

JSONAppendExpr ::= "append" "json" ExprSingle "into" ExprSingle

ItemType ::= -- everything so far --
  | JSONTest
  | StructuredItemTest

JSONTest ::=
    JSONItemTest
  | JSONObjectTest
  | JSONArrayTest

StructuredItemTest ::= "structured-item" "(" ")"
JSONItemTest ::= "json-item" "(" ")"
JSONObjectTest ::= "object" "(" ")"
JSONArrayTest ::= "array" "(" ")"
```

Implementation in Zorba

The present specification was implemented in the Zorba XQuery engine, with the following differences:

- Zorba supports a specific simple object union syntax (for building objects with a dynamic number of pairs): {| Expr? |} corresponds to jn:object(Expr?) in this specification.
- Zorba supports a specific accumulation syntax (for building objects with a dynamic number of pairs without throwing errors upon pair collision): {[Expr?]} corresponds to jn:accumulate(Expr?) in this specification.
- The empty objects syntax {} is not supported, because it collides with empty blocks in Zorba Scripting. However, {| |} can be used instead (see above).
- Zorba supports collections and indices containing JSON items as part of its data definition framework.
- Zorba ships some of the library functions in the jn: module (jn:members, jn:flatten). The others will follow in the next release.

Appendix A. Revision History

Revision 1-3 **Tue Aug 13 2013** **Ghislain Fourny** *g@28.io*

Added more examples for library functions.
Completed signature of keys() and size() with return types.
Simplified code in libjn:remove-keys() and libjn:project().
Moved all fixes for JSON vs. JDM lexical representation mismatch to the serialization layer.

Revision 1-2 **Thu Aug 8 2013** **Ghislain Fourny** *g@28.io*

Fixed the signatures of jn:keys() and jn:members() to accept parameters of type item()*. They were forgotten in the last update. Updated the description accordingly and updating examples with more general sequences.
Fixed some typos.
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 1-1 **Mon Jul 22 2013** **Ghislain Fourny** *g@28.io*

Reintroduced the parameter cast in object lookup (to a string) or array lookup (to an integer) following feedback. The general semantics of object/array lookup with unary dynamic function calls (implicit iteration on mixed sequences, empty sequence in case of mismatch) is mostly untouched, but was significantly reformulated.
Introduced object key listing and array unboxing with 0-ary dynamic function call syntax: $o() (=jn:keys($o)) $a() (=jn:members($a)). It works on mixed sequences in the same way as object and array lookup.
Removed the function jn:is-null() because it is redundant with the "eq" and "instance of" operators which support null.
Removed the function jn:object() and introduced the equivalent {| |} syntax (like in the JSONiq core).
Relaxed the functions libjn:descendant-objects, libjn:descendant-pairs, libjn:values, libjn:flatten, libjn:project to accept any sequence of items. The behavior on mixed sequences is consistent with object and array lookup. Also introduced new function libjn:descendant-arrays and relaxed jn:size to accept the empty sequence..
Fixed roundtrippability of decimals and doubles in jn:encode-for-roundtrip (decimals without dot were roundtripped to integers, etc).
Fixed corner cases in JSON number serialization (trailing zeros, absence of digits, leading +, which are allowed in the JDM but not in the JSON syntax).

Revision 0.4.45 **Mon May 27 2013** **Ghislain Fourny** *g@28.io*

Relaxed object and array selection to work on any mixed sequences. In case of mismatch (i.e., (i) integer on object or atomic, or (ii) string on array or atomic), an empty sequence is now returned. No more cast is done.

Revision 0.4.44 **Thu May 8 2013** **Ghislain Fourny** *g@28.io*

Correcting various typos, errors and formatting errors, mostly in the examples (no change in semantics).

Revision 0.4.43 **Thu Apr 4 2013** **Ghislain Fourny** *ghislain.fourny@28msec.com*

The product was renamed to "JSONiq Extension to XQuery".
The semantics of arithmetics was modified to raise an error in case of a null operand.
The semantics of value comparison was modified to be compatible with null operands (null eq null, and null lt any other atomic item).

Revision 0.4.42 **Thu Sep 27 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

The hybrid output method now serializes XML nodes that are found inside objects or arrays to JSON strings.

Revision 0.4.41 **Tue Sep 25 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Extended fn:string to raise an error for JSON items. Added JNTY0024.
Made definition of roundtripping functions more precise (for the case where there is a top-level item which is a JSON native value).

Revision 0.4.40 **Mon Sep 24 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Renamed JNTY0020 and added JNTY0023.
Changed default of encode-for-roundtrip to omit XML declaration.
Corrected signature of roundtripping functions to allow for full recursion.

Revision 0.4.39 **Fri Sep 14 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Qualified names are now treated specially in jn:encode-for-roundtrip and jn:decode-from-roundtrip.

Revision 0.4.38 **Thu Sep 13 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Further specified error handling for key (atomization error, cast error) in pair constructor.
Fixed appending expression syntax to be in sync with insert syntax.
Object and array selectors are now atomized and cast to xs:string and xs:integer respectively.
The alternate object insertion syntax supports several pairs.

Revision 0.4.37 **Tue Sep 11 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Fixed error code of jerr:JNSE0022.
Moved libjn:members to jn:members (builtin).

Corrected example query for jn:encode-for-roundtrip and excluded NaN/infinite from native JSON mapping.

Revision 0.4.36 **Mon Sep 10 2012** **Ghislain Fourny** ***ghislain.fourny@28msec.com***

Changed error code to err:XPTY0004 if a key in a pair constructor cannot be atomized and cast to a string.
Further specified error-handling for jn:parse-json. Changed default to accepting multiple items.
Simplified semantics for null in comparison/arithmetics. Introduced jn:is-null().
Introduced a lightweight syntax for inserting a single pair into an object (optional curly braces).
Default output method is now JSON-XML-hybrid (also for fn:trace).
Made changes to sequence normalization orthogonal to the output method used. Introduced error codes.

Revision 0.4.35 **Fri Sep 07 2012** **Ghislain Fourny** ***ghislain.fourny@28msec.com***

The json insert syntax was simplified and relaxed.

Revision 0.4.34 **Thu Sep 06 2012** **Ghislain Fourny** ***ghislain.fourny@28msec.com***

Relaxed condition that an expression computing a value (in a pair constructor, an insert expression or a replace expression) must return a single item. If no item, null is used instead, if more than one item, an array is built to wrap the items. Errors jerr:JNTY0002 and jerr:JNUP0017 disappear.
jn:parse-json-sequence was merged with jn:parse-json (using an object to transmit parameters).
jn:encode-for-roundtrip and jn:decode-from-roundtrip also now use an object to transmit parameters.
A semantics is now specified for arithmetic operations involving null (null is returned). The semantics of value and general comparison involving nulls was updated.

Revision 0.4.33 **Tue Sep 04 2012** **Ghislain Fourny** ***ghislain.fourny@28msec.com***

Corrected typos. Capitalized JSON and XML in hybrid output method.
Introduced jn:parse-json-sequence to support roundtripping sequences.

Revision 0.4.32 **Mon Sep 03 2012** **Ghislain Fourny** ***ghislain.fourny@28msec.com***

Split the serialization roundtripping facility into a much simpler specification of the JSON output method and two functions in the builtin module. Only one serialization parameter (specifying whether multiple items are allowed is left).
jerr:JNSE0013 disappears.

Revision 0.4.31 **Fri Aug 31 2012** **Ghislain Fourny** ***ghislain.fourny@28msec.com***

Sequences beginning with an object or array now have an effective boolean value, which is true.

jsoniq-boolean-and-null-literals is set to "yes" by default.

Revision 0.4.30 **Thu Aug 30 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Significantly rewrote the serialization section to take feedback received into account: (i) some serialization parameters were renamed (ii) If jsoniq-roundtrip-extensions is set to no, then non-JSON atomic values are automatically serialized to the JSON literal that makes the most sense (iii) the json-xml-hybrid output method was introduced, mostly meant as a default output method for an engine for convenience.

Revision 0.4.29 **Tue Aug 28 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Added function jn:null.
Specified semantics for casting to/from jn:null.
Specified semantics for value and general comparisons involving JSON nulls.

Revision 0.4.28 **Wed Aug 15 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Added namespace for library functions.

Revision 0.4.27 **Mon Aug 13 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Renamed libjn:accumulator-object-union as libjn:accumulate. Corrected some typos in library function code. Completed paragraph on implementation.

Revision 0.4.26 **Fri Aug 10 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Corrected and moved a dynamic object construction example to lib:object().

Revision 0.4.25 **Wed Aug 8 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Added a chapter about the implementation in Zorba.

Revision 0.4.24 **Tue Aug 7 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Fixed array update grammar rules.

Revision 0.4.23 **Thu Jul 31 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Changed JNDY0018 to JNTY0018.
Added more details on JSONiq string support.

Revision 0.4.22 **Thu Jul 24 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Objects and arrays are no longer matching any function types.
Updated type hierarchy picture.

Revision 0-21 **Thu Jul 24 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Corrected typos (jn:null, object constructor grammar in grammar section, residual {| |}s).
Added update features in the introductive list.

Revision 0-20 **Thu Jun 21 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Added an error code checking for single item for insert/replace expressions.
Corrected typos.
Changed JNTY0007 to JNUP0007 and JNTY0008 to JNUP0008.

Revision 0-19 **Thu Jun 11 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Added error code if the selector cannot be resolved when creating an update primitive.
Updated JSONiq value serialization examples to the right double lexical value of infinity.

Revision 0-18 **Thu May 24 2012** **Ghislain Fourny** *ghislain.fourny@28msec.com*

Pairs are no longer exposed as items. There are now just singleton objects (but the word pair is still used in this spec for a string/value pair);
Update primitives are decoupled from a new update syntax;
Object and array selectors return values;
Dynamic invocation on sequence of functions/arrays/objects;
Minimalistic builtin functions. Still, some functions defined for convenience (relational algebra projection, intersection, ...);
Atomization and EBV raise errors or objects and arrays;
There are only one serialization method: json with three new serialization parameters;
Boolean and null literals can be activated with an option;
Multiple items can be serialized according to serialization parameter settings.

Revision 0-17 **Thu Dec 22 2011** **Jonathan Robie** *jonathan.robie@gmail.com*

Eliminated Array Pairs.
Renamed "member accessors" => "member selectors"
Changed semantics of the empty member selector, which now returns all Objects at the top level or below.
Changed signature of json:delete() to always require the Array or Object as a parameter. Pairs no longer point to the Array or Object that contain them.
Changed namespaces used in the document. Functions and errors have separate namespaces.
Functions used to implement operators and JDM accessors are abstract, and have a prefix that is not bound to a namespace.
Removed isnull() function - compare to null insteadl

Removed whitespace requirement for ":" in PairConstructor, require the operands of a PairConstructor to be AdditiveExprs.
Specified that object lookup uses the codepoint collation.

Revision 0-16 Tue Oct 11 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Eliminated typed-value() and string-value() accessors for Pair.
Redefined atomization to use unboxing; defined type value and string value in terms of unboxing.
Changed signatures of many functions to make them more precise.
Fixed "tuesday" query - result is now "day": "Monday".
Introduced "structured items" base class for nodes and JSON Items.
Added StructuredItemTest.
Removed string arguments from JSONPairTest, JSONObjectPairTest, JSONArrayPairTest.
Specified update primitives.
Made signatures of update primitives more precise.
Stated that NaN and Infinity are instances of **`xs:double`**.
Clarified order of results for () on Arrays.
Changed "unordered" to implementation-defined order.
Minor editorial changes.

Revision 0-15 Thu Sep 22 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Pairs now have typed-values again. No other JSONiq Item does.
Unboxing pairs is now part of atomization.
In the XQ-- subset of JSONiq, atomizing a Pair is defined as unboxing it. The remaining steps of atomization are irrelevant, because XML values do not occur.
Unboxing pairs is also done for the value expression of Array Constructors and Object Constructors. These do not atomize.
The content expressions of element constructors unbox Pairs and flatten arrays. (In other content expressions, atomization is done, so explicit unboxing is not needed.)
Removed dangling references to context items in two function descriptions.
Changed erroneous KindTest to ItemType in several sentences.
Added a footnote clarifying that the identity of JSON Item is used only in PULs.
Changed name of json() to json-doc()
Changed signature of json-doc(), parse-json() to return json-item()?
Put public functions in the **`json`** namespace.
Stated that string serialization uses JSON escape conventions.
Stated that serialization of XML as string content uses serialization parameters and standard serialization.

Revision 0-14 Mon Sep 19 2011 **Jonathan Robie**
jonathan.robie@gmail.com

JSONiq Items no longer have typed values.
Atomization is now defined only for Pairs, which are now unboxed before atomization, then treated as any other value.
In content sequences, Pairs are unboxed, Arrays are flattened, and Objects still raise errors.
Extra-grammatical constraint added: whitespace must occur before and after a ":".
Explicitly stated that a standalone Pair raises an error when serializing.
If a Pair occurs in the value expression of a Pair Constructor, it is unboxed.
Defined the flatten() function to flatten arrays recursively.

Defined serialization for **`json:null`**.

Revision 0-13 Fri Sep 2 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Refers to XQ++ and XQ-- grammars on the JSONiq site.
Tutorial is now a separate document.
Data Model appendix is now part of the main text.
Member accessors are defined independently of dynamic function invocation.
Arrays have a typed value again.
Added type hierarchy diagram.
JSON Item tests are ItemTests, no longer KindTests.
Renamed node types and item tests for consistency, changing terminology in the text to match.
Added item tests for all JSON Item types.
Added isnull() function.
Moved updates out of an appendix and into the main text.

Revision 0-12 Mon Aug 22 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Neither Object nor Array have typed values now.
Coined the term "member accessors".
New section calls out semantics of Pair.
Filled in type signatures of functions, added size().
Added Group By queries to Sample Queries.
Moved Satellites query to Sample Queries.
General reorganization, editorial changes.

Revision 0-11 Fri Aug 12 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Added updating functions, revised update example to use them.

Revision 0-10 Wed Aug 10 2011 **Jonathan Robie**
jonathan.robie@gmail.com

New representation of objects, arrays, pairs in the data model. No longer nodes or function items.
Changed representation of nulls.
Navigation primitives now return pairs, not values.

Revision 0-09 Tue Aug 9 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Significantly changed data model. Pairs now serve as the container for values, in both Arrays and Objects. PULs will now need to be defined for pairs, objects, or arrays, not at the level of a single value.
Changed to new path syntax.

Revision 0-07 Fri Jul 15 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Removed special case positional filters for arrays.

Removed special case conversion of sequences to arrays in object values.
Removed extraneous constraint for objects in insert before / after.

Revision 0-06 Thu Jul 14 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Added status section to emphasize that we are in very early stages.
Changed positioning in the introduction.
Added an update example.
Added an appendix listing changes needed for the XQuery Update Facility.
Added navigation primitives; defined navigation functions in terms of these primitives.

Revision 0-05 Wed Jul 06 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Serialization now uses JSONiq values, with a type and a serialized string, for things that JSON can not represent directly.
Changed the verb used in an example.

Revision 0-04 Tue Jul 05 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Revised author list.
Added explicit JDM representation of nulls.
Changed definition of 'value()' - it now returns the empty sequence rather than a type error for nodes that have no value, and the set of nodes for which it is defined has changed.
Added return type to times-ten() function, showed function call.
Removed superfluous space from svg example.
Changed definition of serialization.
Fixed stock holdings example.
Items are now copied into arrays or objects, rather than using references to nodes in existing hierarchies.

Revision 0-03 Mon Jun 27 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Removed references to XQ--, XQ++, which are not yet well specified.
Finished simple value nodes for XQuery atomic values.
Reworked typed value of nodes, added examples and further explanation to "Combining XML and JSON".
General reorganization and rewriting of introduction.
Added "JSON views in middleware" example.
Added "JSON Arrays to HTML Tables" example.
Added "Transforming to SVG" example.
Added KindTests, with an example.
Added section on combining XML, JSON.
Added extended example to show ramifications of atomization.
Added superficial, hand-wavy section on serialization. Needs discussion / fleshing out.
Reorganized section on navigating JSON objects and arrays.

Revision 0-02 Fri Jun 24 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Clarified datatypes used for numeric types.
Fixed typo in JSONPair grammar.
Added text and an example to clarify use of child axis for JSON Pair Nodes.
Added value() for pair nodes.
Added json() function, similar to XQuery's doc() function.
Added parse-json() function, similar to XQuery's parse-xml() function.
Started a section on combining XML and JSON. Not done yet.
Defined atomization for simple value nodes.

Revision 0-01 Thu Jun 23 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Specified filter expressions for Array nodes.
Added XDM mapping, in an appendix.
Clarified typed-value, string-value for arrays and objects.
NameTest no longer matches Pair Nodes. The differences between JSON names and XML names caused problems because (1) the characters allowed in a NameTest and a JSON string differ, and (2) an unprefixed name in a Nametest is in the default namespace, all JSON strings are in no namespace.
Array constructors now allow empty arrays; Object constructors now allow empty objects.
Made JSONPair available as a primary expression.
Added Stellarium example to introduction.
Incorporated navigation functions based on this week's feedback.

Revision 0-00 Fri Jun 17 2011 **Jonathan Robie**
jonathan.robie@gmail.com

Imported spec into publican.

Index