JSONiq

The SQL of NoSQL

Ghislain Fourny

JSONiq: The SQL of NoSQL

by Ghislain Fourny

Abstract

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. As a result, JSONiq is an expressive and highly optimizable language to query and update any kind of JSONiq store or resource. It enables developers to leverage the same productive high-level language across a variety of NoSQL products. This book gives a complete introduction to the JSONiq language. It does so by giving examples for all types of expressions and functions. Those examples can be immediately used because they work standalone, which allows the interested reader to start diving into the language.

Table of Contents

Chapter 1. Introduction

The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision.
—Niklaus Wirth

NoSQL - Why Are Relational Databases Not Good Enough?

Relational databases have existed for decades. The entity-relationship model is very powerful and with it, it is possible to model almost any structured data. SQL is the widely accepted standard for these databases. It supports the relational algebra operators like join, project, select, filter.

In the last decade, several companies saw the amount of data they needed to store and handle increase dramatically. They soon encountered problems scaling up and out. In his foreword on the "MongoDB Definitive Guide," Jeremy Zawodny explained it convincingly: once you add more replicas and shards, you realize you are stuck in the original schema you designed unless you invest considerable effort.

In order to solve this issue, a new generation of data stores appeared. They often share the same set of design principles:

- The lines of a relational table are replaced with hierarchical data (semi-structured data, aka trees), while tables become collections of trees.
- These trees are primarily associated with and indexed by an ID or a key.
- Schemas are not mandatory, i.e., trees within a collection need not share the same structure (heterogeneity).
- Some data stores see a tree as a kind of black box (key/value stores) while some other data stores use XML (like eXist) and more recently JSON (like MongoDB) syntax to represent trees.

These new technologies are often referred to as "NoSQL."

Why JSONiq?

NoSQL has a very broad meaning and, while the general principles are similar between data stores, each data store has a specific format for the values (or trees) and a query language tailored for the data store.

JSONiq was developed with the idea that many data stores share the same design principles (e.g., collections of trees) so that it should be possibly to query them in a unified and portable way.

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 results 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. Neither does it specify

whether an algorithm is executed in main memory or whether it is executed 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 language for nested and heterogeneous data.

How to Run the Queries in This Book?

Our first implementation of JSONiq was done in the Zorba NoSQL processor, which is developed jointly between Oracle, 28msec, and the FLWOR Foundation. The home page is *http://zorba.io/* and a sandbox is available on *http://try.zorba.io/*. You can run most of the queries shown in the examples of this book in this sandbox (not the ones accessing collections).

28msec provides a platform called 28.io, which is specifically tailored for executing JSONiq queries against an existing MongoDB database. You can run all example queries in the Try-It-Now sandbox at *http://28.io/*, in which the collections *faqs* and *answers* are prepopulated with lots of data and additional sample queries.

If you obtained this book shortly after its publication, you should be aware that the array unboxing syntax may not be released yet, as it is recent. If array unboxing does not work, try `$a()` instead of `$a[]` and `$a(2)` instead of `$a[][2]`.

Acknowledgements

The design and implementation of JSONiq is a team effort involving Dana Florescu (Oracle), Jonathan Robie (EMC), Matthias Brantner (28msec), Markos Zaharioudakis (Oracle), Till Westmann (Oracle) and myself (28msec).

This book was carefully reviewed by Matthias Brantner, Federico Cavalieri (28msec) and Paul J. Lucas (28msec). Many thanks to them!

A significant part of the introduction ("Why JSONiq?") was written by Dana Florescu.

Part I. JSON and the JSONiq Data Model

Chapter 2. The JSON Syntax

The JSONiq query language was specifically designed for querying and processing JSON.

As stated on its home page *http://www.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.”

JSON itself is only about syntax: a string may or may not match the JSON grammar. If it does, then it is well-formed JSON. The JSON syntax is made of the following building blocks: objects, arrays, strings, numbers, booleans and nulls. Let us begin with a quick overview of all these building blocks.

JSON Strings

Strings are double-quoted. To put it simply, they are sequences of Unicode characters with absolutely no restriction:

```
"foo",
"What NoSQL solutions are out there?"
```

However, syntactically, some of these characters must be escaped with backslashes (escape sequence). This includes double quotes, escaped as \" -- because otherwise they could be confused with the end of a string -- and backslahes themselves, escaped as \\ -- because otherwise you would not know if you mean a backslash character, or if you are escaping the following character.

```
"What \"NoSQL\" solutions are out there?"
```

Finally, all Unicode control characters (null, new line, form feed, delete...) are not allowed directly and must be built with an escape sequence. Any Unicode character, including control characters, can be built with \u followed by the four hexadecimal digits that identify it within Unicode. The most frequent control characters even have their own shortcuts: \n (new line), \t (tab), \r (carriage return), \b (backspace), \f (form feed). The slash can also be obtained with \/, although it is fine too if it appears alone. This is useful in JSON-hosting environments where slashes are special.

```
"What \"NoSQL\" solutions are out there:\n"
"MapReduce\u000AMongoDB\n\u0085"
```

JSON Numbers

Numbers cover the entire decimal space. There is no range restriction. Although there is no formal distinction in JSON, numbers can be grouped into three subcategories. These subcategories play an important role in JSONiq.

- Integers, possibly with a negative sign and not beginning with a leading 0 (except 0 itself):

  ```
  0
  9
  42
  -96
  123456789012345678901234567890123456789012345
  ```

- "Plain" decimals, with a dot, both followed and preceded by at least by one digit (no leading dot):

  ```
  0.3
  9.6
  42.2346902834
  -96.01345023400
  ```

- Decimals in scientific notation, i.e., a plain decimal followed by an E (case does not matter) and by a power of ten (an integer with an optional sign):

  ```
  0.3e0
  9.6E+24
  42.2346902834e-2
  -96.01345023400E-02345
  ```

JSON Booleans

Booleans cover the two logical truth values true and false, unquoted. There is not much more to say about them...

```
true
false
```

JSON Null

Null is a special value that can be used to denote the absence of value.

```
null
```

JSON Objects

Objects are unordered sets of key/value pairs. A key is any JSON string as described above. A value is any JSON building block.

According to the JSON RFC, keys (the strings) should be unique within the same object -- and JSONiq does consider them unique.

You can see in the following examples that values can be also nested objects or arrays.

```
{
  "_id" : "511C7C5C9A277C22D138802D",
  "question_id" : 4419499,
  "last_edit_date" : "2012-12-17T00:02:31",
  "creation_date" : "2010-12-11T23:15:19",
  "last_activity_date" : "2012-12-17T00:02:31",
  "score" : 15,
  "accepted_answer_id" : 4421601,
  "title" : "MySQL and NoSQL: Help me to choose the right o
ne",
  "tags" : [ "php", "mysql", "nosql", "cassandra" ],
  "view_count" : 3972,
  "owner" : {
    "user_id" : 279538,
    "display_name" : "cedivad",
    "reputation" : 430,
    "user_type" : "registered",
    "profile_image" : "http://www.gravatar.com/avatar/b77fa
dd2ba791134ac40a9c184be1eda?d=identicon&amp;r=PG",
    "link" : "http://stackoverflow.com/users/279538/cedivad
",
    "accept_rate" : 74
  },
  "link" : "http://stackoverflow.com/questions/4419499/mysq
l-and-nosql-help-me-to-choose-the-right-one",
  "is_answered" : true
}
```

In the NoSQL world, top-level JSON objects are often referred to as JSON documents.

Chapter 3. The JSONiq Data Model

Having a JSON document as pure syntax is not very useful in itself, except to send it over a network or to store it in a document store of course. To make use of it in a database or in other processing environments, you need to bring it to a higher level of abstraction and give semantics to the building blocks. This is what a Data Model is for.

We now introduce the JSONiq data model.

Let us begin with some good news first: the JSON syntax that we have just introduced is a subset of JSONiq. Concretely, this means that any of these syntactic JSON building blocks can be copy-and-pasted, and executed as a JSONiq query. The output will be the counterpart of this JSON building block in the Data Model. So, if you are familiar with JSON, then you already know some JSONiq.

JSONiq Values: Items and Sequences

In JSONiq, the JSON building blocks described in the former section, on a more abstract level, are referred to as items. JSONiq manipulates sequences of these items. Hence, a JSONiq value is a sequence of items. So, in particular, a JSONiq query returns sequences of items. Actually, even inside a JSONiq query, sequences of items are passed around between the JSONiq building blocks internal to a query (called expressions).

Let us copy-and-paste a JSON Object and execute it as JSONiq:

Example 3.1. A sequence of just one item.

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

Results:

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

The above query generates a sequence of one item, an object item in this case. The result displayed above is the output of this query when run with the Zorba query processor, which is one of the JSONiq implementations.

Commas are all you need to begin building your own sequences. You can mix and match!

Example 3.2. A sequence of various items.

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

Results:

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

There are three golden rules about sequences that are useful to keep in mind.

Rule #1: Sequences are flat and cannot be nested. This makes streaming possible, which is very powerful.

Example 3.3. Sequences are flat.

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

Results:

```
"foo"
2
true
4
null
6
```

Rule #2: A sequence can be empty. The empty sequence can be constructed with empty parentheses.

Example 3.4. The empty sequence.

```
()
```

Results:

Rule #3: A sequence of just one item is considered the same as this item itself. Whenever we say that an expression returns or takes one item, we really mean that it takes a singleton sequence of one item.

Example 3.5. A sequence of one item.

```
("foo")
```

Results:

```
"foo"
```

JSONiq classifies the items mentioned above in three categories:

- Objects: the counterparts of the syntactic JSON objects.
- Arrays: the counterparts of the syntactic JSON arrays.
- Atomics: the counterparts of JSON strings, JSON numbers, JSON booleans and JSON nulls - but with a very rich type system which includes dates, for example.

Objects

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

Each pair consists of an atomic of type *string* and of an item which can be in any category.

No two pairs have the same name. Because of this, the word *field* is also used to refer to pairs.

Arrays

An array represents a JSON array: an ordered list of items -- items in any category.

An array can be seen as a sequence that is wrapped in one single item. And since an array is an item, arrays can nest -- like in JSON.

Atomics

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

JSONiq defines many useful builtin atomic types. For now, let us introduce those that have a JSON counterpart. 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. Here is a little appetizer that showcases constructing a date and a duration (365 days), and adding them.

Example 3.6. Atomics with the types date and dayTimeDuration.

```
date("2013-06-21") + dayTimeDuration("P365D")
```

Results:

```
"2014-06-21"
```

Chapter 4. The JSONiq Type System

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.

Like in Java or C++, it is possible to create a variable with a given static type:

Example 4.1. Specifying a type.

```
let $x as integer := 16
return $x * $x
```

Results:

```
256
```

Like in JavaScript, it is possible to create a variable without explicitly giving any static type. JSONiq is still strongly typed, so that you will be told if there is a type inconsistency or mismatch in your programs.

Example 4.2. Not specifying a type.

```
let $x := 16
return $x * $x
```

Results:

```
256
```

Variables are explained in the section called “Variables” in Chapter 10, *FLWOR Expressions* more in details.

JSONiq supports types at the sequence level. They are called sequence types, and the syntax for designing types is called the sequence type syntax. The type "integer" that was shown in

Example 4.1, “Specifying a type.” matches singleton sequences of one atomic item of type integer.

We say that a sequence matches a sequence type (or that a sequence type matches a sequence) if the sequence is in the value space of the sequence type. Since an item is a particular (singleton) sequence, we also can say that an item matches an item type or conversely.

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. But it is not forbidden for the processor to be smart and warn you if it can detect that a type issue can arise at runtime.

There are many JSONiq expressions (cast, instance of, ...) which perform type operations and that make use of the sequence type syntax. In the remainder of this section, we will introduce sequence types using an "instance of" expression that returns true or false depending on whether or not the type on the right side is matched by the value on the left side -- like in Java.

Example 4.3. The instance of operator.

```
16 instance of integer
```

Results:

```
true
```

Item Types

Atomic Types

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 that 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 that matches *integer*.

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

  All decimal literals build an atomic that matches *decimal*.

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

  All double literals build an atomic that matches *double*.

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

  All boolean literals build an atomic that matches *boolean*.

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

  All null literals build an atomic that matches *null*.

- *atomic*: all atomic types.

  All literals build an atomic that matches *atomic*.

Example 4.4. Atomic types

```
16 instance of integer,
16 instance of decimal,
16.6 instance of decimal,
16.6e10 instance of double,
"foo" instance of string,
true instance of boolean,
null instance of null,
"foo" instance of atomic
```

Results:

```
true
true
true
true
true
true
true
true
```

JSONiq also supports further atomic types, which were borrowed from XML Schema 1.1.

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.

Atomic items that have these builtin atomic types can only be built with a constructor -- again similar to JavaScript.

Example 4.5. Further builtin atomic types.

```
date("2013-06-18") instance of date,
dateTime("2013-06-21T05:00:00Z") instance of dateTime,
time("05:00:00") instance of time,
long("1234567890123") instance of long
```

Results:

```
true
true
true
true
```

JSON Item Types : Object Types and Array Types

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

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

Atomics do not match *json-item*.

Example 4.6. Further builtin atomic types.

```
{ "foo" : "bar" } instance of object,
{ "foo" : "bar" } instance of json-item,
{} instance of object,
[ 1, 2, 3, 4 ] instance of array,
[ 1, 2, 3, 4 ] instance of json-item
```

Results:

```
true
true
true
true
true
```

The Most General Item Type.

All items match the item type *item*.

Example 4.7. The most general item type: item.

```
{ "foo" : "bar" } instance of item,
[ 1, 2, 3, 4 ] instance of item,
"foo" instance of item,
42 instance of item,
false instance of item,
null instance of item
```

Results:

```
true
true
true
true
true
```

```
true
```

Sequence Types

All sequences match the sequence type *item**.

Example 4.8. The most general sequence type: item*.

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

Results:

```
true
true
true
```

But sequence types can be much more precise than that. In general, a sequence type is made of an item type, as presented above, followed by an occurrence indicator among the following:

- * 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).

Example 4.9. Further sequence types.

```
( { "foo" : "bar" } , {} ) instance of object*,
() instance of object*,
( [ 1, 2, 3 ] , {} ) instance of json-item+,
[ 1, 2, 3 ] instance of array?,
() instance of array?,
"foo" instance of string
```

Results:

```
true
true
true
true
true
true
```

There is also a special type that matches only empty sequences, denoted () as well:

Example 4.10. Empty sequence type: ()

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

Results:

```
true
```

Part II. Construction of Items and JSON Navigation

Chapter 5. Construction of Items

As we just saw, the items (objects, arrays, strings, ...) mentioned in Chapter 2, *The JSON Syntax* are constructed exactly as they are constructed in JSON. In a way, any JSON building block is also a well-formed JSONiq query which just "returns itself" (more precisely: its counterpart in the JSONiq Data Model).

Atomic Literals

String Literals

The syntax for creating strings is identical to that of JSON. No surprise here. JSON's backslash escaping is supported, and like in JSON, double quotes are required and single quotes are forbidden.

Example 5.1. String literals.

```
"foo",
"This is a line\nand this is a new line",
"\u0001",
"This is a nested \"quote\""
```

Results:

```
"foo"
"This is a line
and this is a new line"
"&#x1;"
"This is a nested "quote""
```

Number Literals.

The syntax for creating numbers is identical to that of JSON.

Example 5.2. Number literals (integer, decimal and double literals)

```
42,
3.14,
-6.022E23
```

Results:

```
42
3.14
-6.022E23
```

Well, not quite. Actually, JSONiq allows a more flexible superset. In particular:

- leading 0s are allowed
- a decimal literal can begin or end with a dot
- a number may begin with a + sign

Example 5.3. A more general literal syntax.

```
042,
.1415926535,
42.,
+6.022E23
```

Results:

```
42
0.1415926535
42
6.022E23
```

Remember 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 either.

Boolean and Null Literals

There is not much to say actually -- boolean literals build boolean atomics, the null literal builds a null atomic, so no worries here, the world is in order.

Example 5.4. Boolean and null literals.

```
true,
false,
null
```

Results:

```
true
false
null
```

Object Constructors

The syntax for creating objects is also 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 5.5. Object constructors.

```
{},
{ "foo" : "bar" },
{ "foo" : [ 1, 2, 3, 4, 5, 6 ] },
{ "foo" : true, "bar" : false },
{ "this is a key" : { "value" : "a value" } }
```

Results:

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

```
{
  "foo" : [ 1, 2, 3, 4, 5, 6 ]
}
{
  "foo" : true,
  "bar" : false
}
{
  "this is a key" : {
    "value" : "a value"
  }
}
```

Again, JSONiq is more flexible here. Like in JavaScript, if your key is simple enough (like alphanumerics, underscores, dashes, these kinds of things), you are welcome to omit the quotes. The strings for which quotes are not mandatory are called *unquoted names*. This class of strings can be used for unquoted keys, but also in later sections for variable and function names, and for module aliases.

Example 5.6. Object constructors with unquoted keys.

```
{ foo : "bar" },
{ foo : [ 1, 2, 3, 4, 5, 6 ] },
{ foo : "bar", bar : "foo" },
{ "but you need the quotes here" : null }
```

Results:

```
{
  "foo" : "bar"
}
{
  "foo" : [ 1, 2, 3, 4, 5, 6 ]
}
{
  "foo" : "bar",
  "bar" : "foo"
}
{
  "but you need the quotes here" : null
}
```

Array Constructors

The syntax for creating arrays is identical to that of JSON (do you sense a growing feeling that we are repeating ourselves? But it feels so good to say it): square brackets, comma separated values.

Example 5.7. Empty array constructors.

```
[],
[ 1, 2, 3, 4, 5, 6 ],
[ "foo", [ 3.14, "Go" ], { "foo" : "bar" }, true ]
```

Results:

```
[  ]
[ 1, 2, 3, 4, 5, 6 ]
[ "foo", [ 3.14, "Go" ], { "foo" : "bar" }, true ]
```

Square brackets are mandatory. Things can only be pushed so far.

Composing Constructors

Of course, JSONiq would not be very interesting if all you could do is copy and paste JSON documents. So now is time to get to the meat.

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*. An expression is the JSONiq building block. You already know some (literals, constructors, comma, cast, instance of) and plenty more will be introduced in the next part (arithmetics, logic, comparison, if-then-else, try-catch, FLWORS that allow you to join, select, group, filter, project, stream in windows, ...)

In order to illustrate composability, the following examples use a few of the many operators you can use:

- "to" for creating sequences of consecutive integers,
- "||" for concatenating strings,

- "+" for adding numbers,
- "," for appending sequences (yes, you already know this one).

So here we go.

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

Example 5.8. Composable array constructors.

```
[ 1 to 10 ],
[ "foo" || "bar", 1 to 3, 2 + 2 ]
```

Results:

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

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

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

Example 5.9. Composable object keys.

```
{ "foo" || "bar" : true },
{ 1 + 1 : "foo" }
```

Results:

```
{
  "foobar" : true
}
{
  "2" : "foo"
}
```

And do not worry about the value expression: if it 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 5.10. Composable object values.

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

Results:

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

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

Example 5.11. Merging object constructor.

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

Results:

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

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

Chapter 6. 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.

We now introduce collections, because collections are perfect to illustrate the JSON navigation syntax which will be introduced in the next section.

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 book.

We will just assume that there is a function named collection() that returns all objects associated with the provided collection name.

Example 6.1. Getting all objects from a collection.

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

Results:

```
{
  "question" : "What NoSQL technology should I use?"
}
```

Collections Used Throughout This Book

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

- *collection("one-object")*

  Example 6.2. The object in the one-object collection.

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

Results:

```
{
  "question" : "What NoSQL technology should I use?"
}
```

- *collection("faqs")* - this is a collection of StackOverflow FAQs.

Example 6.3. One object from the faqs collection.

```
collection("faqs")[1]
```

Results:

```
{
  "_id" : "511C7C5C9A277C22D138802D",
  "question_id" : 4419499,
  "last_edit_date" : "2012-12-17T00:02:31",
  "creation_date" : "2010-12-11T23:15:19",
  "last_activity_date" : "2012-12-17T00:02:31",
  "score" : 15,
  "accepted_answer_id" : 4421601,
  "title" : "MySQL and NoSQL: Help me to choose the right o
ne",
  "tags" : [ "php", "mysql", "nosql", "cassandra" ],
  "view_count" : 3972,
  "owner" : {
    "user_id" : 279538,
    "display_name" : "cedivad",
    "reputation" : 430,
    "user_type" : "registered",
    "profile_image" : "http://www.gravatar.com/avatar/b77fa
dd2ba791134ac40a9c184be1eda?d=identicon&amp;r=PG",
    "link" : "http://stackoverflow.com/users/279538/cedivad
",
    "accept_rate" : 74
  },
  "link" : "http://stackoverflow.com/questions/4419499/mysq
l-and-nosql-help-me-to-choose-the-right-one",
  "is_answered" : true
}
```

- *collection("answers")* - this is a collection of StackOverflow answers (to the previous FAQs).

Example 6.4. One object from the answers collection.

```
collection("answers")[1]
```

Results:

```
{
  "_id" : "511C7C5D9A277C22D13880C3",
  "question_id" : 37823,
  "answer_id" : 37841,
  "creation_date" : "2008-09-01T12:14:38",
  "last_activity_date" : "2008-09-01T12:14:38",
  "score" : 7,
  "is_accepted" : false,
  "owner" : {
    "user_id" : 2562,
    "display_name" : "Ubiguchi",
    "reputation" : 1871,
    "user_type" : "registered",
    "profile_image" : "http://www.gravatar.com/avatar/00b87
a917ec763c0c051dc6b8c06f402?d=identicon&amp;r=PG",
    "link" : "http://stackoverflow.com/users/2562/ubiguchi"

  }
}
```

Many queries in this book can be directly input into 28.io's try-it-now sandbox, as these collections are preloaded (this is real-world data).

Chapter 7. JSON Navigation

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

JSONiq supports:

- Looking up the value of a field (given its string key) in an object.
- Looking up the item at a given position (integer) in an array.
- Extracing all members of an array as a sequence of items.
- Filtering items from a sequence, retaining only the items that match a given criterium.

Object Navigation

The simplest way to navigate an object is similar to JavaScript. This will work as soon as you do not push it too much: alphanumerical characters, dashes, underscores. The rule for unquoted names is similar to keys in object constructions, and to variable names. The empty sequence is returned if no key is found with the specified name.

Example 7.1. Object lookup.

```
{
  "question" : "What NoSQL technology should I use?"
}.question,
{
  "question" : "What NoSQL technology should I use?"
}.answer
```

Results:

```
"What NoSQL technology should I use?"
```

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 7.2. Lookup on a single-object collection.

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

Results:

```
"What NoSQL technology should I use?"
```

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 any item which is not an object (arrays and atomics) results in the empty sequence.

Example 7.3. Object lookup with an iteration on several objects.

```
({ "foo" : "bar" }, { "foo" : "bar2" } ).foo,
{ "ids" : collection("faqs").question_id }
```

Results:

```
"bar"
"bar2"
{
  "ids" : [ 4419499, 282783, 4720508, 5453872, 6183352 ]
}
```

Example 7.4. Object lookup on non-objects.

```
"foo".foo,
({
  "question" : "What NoSQL technology should I use?"
},
 [ "question", "answer" ],
 { "question" : "answer" },
 "question").question
```

Results:

```
"What NoSQL technology should I use?"
"answer"
```

Of course, unquoted keys will not work for strings that are not unquoted names, e.g., if the field contains a dot or begins with a digit. Then you will need quotes. If you use a more general expression on the right-hand side of the dot, it must always have parentheses.

Example 7.5. Quotes and parentheses for object lookup.

```
{
  "my question" : "What NoSQL technology should I use?"
}."my question",
{
  "my question" : "What NoSQL technology should I use?"
}.("my " || "question")
```

Results:

```
"What NoSQL technology should I use?"
"What NoSQL technology should I use?"
```

The value returned by the right-hand side expression is cast to string. An error is raised upon failure. This value may be the empty sequence, in which case the object lookup also returns the empty sequence.

Example 7.6. Object lookup with a nested expression.

```
{
  "question" : "What NoSQL technology should I use?"
}.(),
{
  "1" : "What NoSQL technology should I use?"
}.(1),
{
  "1" : "What NoSQL technology should I use?"
}."1"
```

Results:

```
"What NoSQL technology should I use?"
"What NoSQL technology should I use?"
```

Variables, or a context item reference, do not need parentheses. Variables are introduced in the section called “Variables”, but here is a sneak peek:

Example 7.7. Object lookup with a variable.

```
let $field := "my " || "question"
return {
  "my question" : "What technology should I use?"
}.$field
```

Results:

```
"What technology should I use?"
```

Array Unboxing

The items in an array (which is an item) can be extracted as a sequence of items with the [] postfix operator.

The argument must be (a singleton sequence of) one array or the empty sequence (in which case the empty sequence is returned as well.

Example 7.8. Array unboxing.

```
[
  "What NoSQL technology should I use?",
  "What is the bottleneck in MapReduce?"
][],

for $a in collection("faqs").tags
return $a[],

()[]
```

Results:

```
"What NoSQL technology should I use?"
"What is the bottleneck in MapReduce?"
"php"
```

```
"mysql"
"nosql"
"cassandra"
"sql"
"database"
"nosql"
"non-relational-database"
"nosql"
"couchdb"
"cassandra"
"redis"
"database"
"full-text-search"
"nosql"
"couchdb"
"riak"
"database"
"view"
"nosql"
"couchdb"
```

Sequence Filtering

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. The predicate expression can use $$ to refer to the item being processed, called the context item.

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

Example 7.9. Predicate expression for picking an item at a given position.

```
(1 to 10)[5],
(
  "What NoSQL technology should I use?",
  "What is the bottleneck in MapReduce?"
)[2]
```

Results:

```
5
"What is the bottleneck in MapReduce?"
```

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 7.10. Predicate expression for filtering.

```
(
  "What NoSQL technology should I use?",
  "What is the bottleneck in MapReduce?"
)[contains($$, "NoSQL")],

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

Results:

```
"What NoSQL technology should I use?"
2
4
6
8
10
```

Array Navigation

Once you know how to unbox an array and to filter a sequence, array lookup comes for free. It feels very much like opening a box of Swiss chocolate and then picking your favorite:

- Unbox the array with [].
- Pick the $i-th item in the sequence using a predicate with an integer [$i].

Example 7.11. Array lookup.

```
[ "question", "answer" ][][2],
{
  questions: [
    "What NoSQL technology should I use?",
    { "faq" : "What is the bottleneck in MapReduce?" }
  ]
}.questions[][2].faq
```

Results:

```
"answer"
"What is the bottleneck in MapReduce?"
```

Part III. JSONiq Expressions

Chapter 8. Basic Operations

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

Construction of Sequences

Comma Operator

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

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

Example 8.1. Comma.

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

Results:

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

```
3
4
5
```

Range Operator

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

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

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

Example 8.2. Range operator.

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

Results:

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

Parenthesized Expressions

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

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

Example 8.3. Empty sequence.

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

Results:

```
25
```

Arithmetics

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

Example 8.4. Basic arithmetic operations with precedence override.

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

Results:

```
8
```

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

Example 8.5. Using basic operations with dates.

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

Results:

```
"P29D"
```

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

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

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

Example 8.6. Basic arithmetic operations with different, but compatible number types

```
2.3e4 + 5
```

Results:

```
23005
```

String Concatenation

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

Example 8.7. String concatenation.

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

Results:

```
"Captain Kirk"
"CaptainKirk"
```

Comparison

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

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

Example 8.8. Equality comparison.

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

Results:

```
true
true
```

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

Example 8.9. Equality and non-equality comparison with null.

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

Results:

```
false
true
true
```

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

Example 8.10. Ordering comparison with null.

```
null lt 1
```

Results:

```
true
```

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

Empty Sequence Behavior

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

Example 8.11. The empty sequence used in basic operations.

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

Results:

Logic

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

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

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

Example 8.12. Conversion to booleans.

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

Results:

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

Propositional Logic

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

Example 8.13. Logics with booleans.

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

Results:

```
true
```

```
true
```

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

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

Example 8.14. Non-determinism in presence of errors.

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

Results:

```
true
```

First-Order Logic (Quantified Variables)

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

Example 8.15. Universal and existential quantifiers.

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

Results:

```
true
true
```

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

Example 8.16. Existential quantifier with type checking.

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

Results:

```
true
```

Builtin Functions

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

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

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

JSONiq supports three sorts of functions:

- Builtin functions: these have no prefix and can be called without any import.
- Local functions: they are defined in the prolog, to be used in the main query. They have the prefix *local:*. Chapter 12, *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 13, *Modules* describes how to define your own modules.

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

Example 8.17. A builtin function call.

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

Results:

```
"foo"
```

```
"bar"
"foobar"
```

Some builtin functions perform aggregation and are particularly convenient:

Example 8.18. A builtin function call.

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

Results:

```
5050
50.5
20
```

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

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

Example 8.19. Calling a function with a sequence.

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

Results:

```
4
```

Chapter 9. Control Flow Expressions

JSONiq supports control flow expressions such as conditional expressions (if then else), switch, and typeswitch. At least the first two should be familiar to any programmer.

Conditional Expressions

A conditional expression allows you to pick the one or the other value depending on a boolean value.

Example 9.1. A conditional expression.

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

Results:

```
{
  "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. The exists() builtin function can be useful to know if a sequence is empty or not.

Example 9.2. A conditional expression.

```
if (null) then { "foo" : "yes" }
          else { "foo" : "no" },
if (1) then { "foo" : "yes" }
       else { "foo" : "no" },
if (0) then { "foo" : "yes" }
       else { "foo" : "no" },
if ("foo") then { "foo" : "yes" }
           else { "foo" : "no" },
if ("") then { "foo" : "yes" }
        else { "foo" : "no" },
if (()) then { "foo" : "yes" }
        else { "foo" : "no" },
if (exists(collection("faqs"))) then { "foo" : "yes" }
                               else { "foo" : "no" }
```

Results:

```
{
  "foo" : "no"
}
{
  "foo" : "yes"
}
{
  "foo" : "no"
}
{
  "foo" : "yes"
}
{
  "foo" : "no"
}
{
  "foo" : "no"
}
{
  "foo" : "yes"
}
```

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

Example 9.3. A conditional expression.

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

Results:

```
{
  "foo" : "yes"
}
```

Switch expressions

Switch expressions are very similar to C++. 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 9.4. A switch expression.

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

Results:

```
"bar"
```

If the provided value is not an atomic, an error is raised (this is also similar to C++).

If the value does not match any of the expected values, the default is used.

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

Example 9.5. A switch expression.

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

Results:

```
"none"
```

The case clauses support composability of expressions as well - an opportunity to remind you about the precedence of the comma.

Example 9.6. A switch expression.

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

Results:

```
"foo"
"1 + 1 is 2"
```

Try-Catch expressions

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

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

Example 9.7. A try catch expression.

```
try { 1 div 0 } catch * { "Caught!" }
```

Results:

```
"Caught!"
```

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

Example 9.8. An error outside of a try-catch expression (failing).

```
let $x := 1 div 0
```

```
return try { $x }
catch * { "Caught!" }
```

Error:

```
division by zero
```

Errors that are detected statically within the try block, for example syntax errors, are still reported statically.

Note that this applies also if the engine is capable of detecting a type error statically, while another engine might only discover it at runtime and catch it. You should keep this in mind, and only use try-catch expressions as a safety net.

Example 9.9. A try catch expression with a syntax error (failing).

```
try { x } catch * { "Caught!" }
```

Error:

```
invalid expression: syntax error, a path expression cannot
begin with an axis step
```

Example 9.10. A try catch expression with a type error (no guarantee of failure or success).

```
try { "foo" + "bar" } catch * { "Caught!" }
```

Results:

```
"Caught!"
```

Chapter 10. FLWOR Expressions

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).

Let us begin with a bit of theory on how they work.

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. It is not to be confused with Java or C++ return statements, as it does not exit or break the loop.

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 -- but first, we need to introduce variable syntax.

Variables

Values can be bound to variables within a certain scope. Variable references always begin with a dollar sign: $foo.

The scope of a variable declared in a FLWOR clause comprises all further clauses of the FLWOR expression up to the return clause.

Variables are immutables, but variable bindings can be hidden with a binding to a variable with the same name.

Variables can be declared by FLWOR expressions as shown in this chapter, but also as global variables (the section called “Global Variables”) or in typeswitch expressions (the section called “Typeswitch Expressions”).

There is a special variable which is called the context item and which is denoted with $$. You already saw it in the section called “Sequence Filtering” in Chapter 7, *JSON Navigation*.

For Clauses

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.

For example, the following for clause:

```
for $x in 1 to 3
...
```

produces the following stream of tuples. The tuples themselves are for explanatory purposes, they are not part of the data model. The syntax is also ad-hoc and is used for illustrating.

```
$x : 1
$x : 2
$x : 3
```

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 display_name FROM answers". $x is bound in turn to each item in the answers collection.

Example 10.1. A for clause.

```
for $x in collection("answers")
return $x.owner.display_name
```

Results:

```
"Ubiguchi"
"Rob Wells"
"Victor Nicollet"
"descent89"
```

```
"JasonSmith"
"JasonSmith"
"JasonSmith"
"JasonSmith"
```

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

Example 10.2. Two for clauses.

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

Results:

```
11
12
13
21
22
23
31
32
33
```

Example 10.3. A for clause with two variables.

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

Results:

```
11
12
13
21
22
23
31
```

```
32
33
```

A for variable is visible to subsequent bindings.

Example 10.4. Two for clauses.

```
for $x in ( [ 1, 2, 3 ],
            [ 4, 5, 6 ],
            [ 7, 8, 9 ] ),
    $y in $x[]
return $y,
for $x in collection("faqs")[size($$.tags) eq 5],
    $y in $x.tags[]
return {
  "id" : $x.question_id,
  "tag" : $y
}
```

Results:

```
1
2
3
4
5
6
7
8
9
{
  "id" : 5453872,
  "tag" : "database"
}
{
  "id" : 5453872,
  "tag" : "full-text-search"
}
{
  "id" : 5453872,
  "tag" : "nosql"
}
{
  "id" : 5453872,
  "tag" : "couchdb"
}
{
  "id" : 5453872,
  "tag" : "riak"
}
```

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

Example 10.5. A for clause with a position variable.

```
for $x at $position in collection("answers")
return {
  "old id" : $x.answer_id,
  "new id" : $position
}
```

Results:

```
{
  "old id" : 37841,
  "new id" : 1
}
{
  "old id" : 37844,
  "new id" : 2
}
{
  "old id" : 4419542,
  "new id" : 3
}
{
  "old id" : 4419578,
  "new id" : 4
}
{
  "old id" : 4720977,
  "new id" : 5
}
{
  "old id" : 5454583,
  "new id" : 6
}
{
  "old id" : 6195094,
  "new id" : 7
}
{
  "old id" : 6210422,
  "new id" : 8
}
```

JSONiq supports joins. For example, the counterpart of "SELECT q.title AS question, q.question_id FROM faq q JOIN answers a ON q.question_id = a.question_id" is:

Example 10.6. A regular join.

```
for $question in collection("faqs"),
    $answer in collection("answers")
    [ $$.question_id eq $question.question_id ]
return { "question" : $question.title,
         "answer score" : $answer.score }
```

Results:

```
{
  "question" : "MySQL and NoSQL: Help me to choose the right
 one",
  "answer score" : 17
}
{
  "question" : "MySQL and NoSQL: Help me to choose the right
 one",
  "answer score" : 1
}
{
  "question" : "Redis, CouchDB or Cassandra?",
  "answer score" : 34
}
{
  "question" : "Full-text search in NoSQL databases",
  "answer score" : 6
}
{
  "question" : "Find CouchDB docs missing an arbitrary field
",
  "answer score" : 0
}
{
  "question" : "Find CouchDB docs missing an arbitrary field
",
  "answer score" : 1
}
```

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 answer for a question. The following query is the counterpart of "SELECT q.title AS question, q.question_id FROM faq q LEFT JOIN answers a ON q.question_id = a.question_id".

Example 10.7. An outer join.

```
for $question in collection("faqs"),
    $answer allowing empty in collection("answers")
    [ $$.question_id eq $question.question_id ]
return { "question" : $question.title,
         "answer score" : $answer.score }
```

Results:

```
{
  "question" : "MySQL and NoSQL: Help me to choose the right
 one",
  "answer score" : 17
}
{
  "question" : "MySQL and NoSQL: Help me to choose the right
 one",
  "answer score" : 1
}
{
  "question" : "The Next-gen Databases",
  "answer score" : null
}
{
  "question" : "Redis, CouchDB or Cassandra?",
  "answer score" : 34
}
{
  "question" : "Full-text search in NoSQL databases",
  "answer score" : 6
}
{
  "question" : "Find CouchDB docs missing an arbitrary field
",
  "answer score" : 0
}
{
  "question" : "Find CouchDB docs missing an arbitrary field
",
  "answer score" : 1
}
```

Where Clauses

Where clauses are used for filtering.

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 q.title as question, q.question_id as id FROM faq WHERE CONTAINS(question, 'NoSQL')".

Example 10.8. A where clause.

```
for $question in collection("faqs")
where contains($question.title, "NoSQL")
return {
  "question" : $question.title,
  "id" : $question.question_id
}
```

Results:

```
{
  "question" : "MySQL and NoSQL: Help me to choose the right
 one",
  "id" : 4419499
}
{
  "question" : "Full-text search in NoSQL databases",
  "id" : 5453872
}
```

JSONiq can do joins with where clauses, too:

Example 10.9. A join with a where clause.

```
for $question in collection("faqs"),
    $answer in collection("answers")
where $question.question_id eq $answer.question_id
return {
  "question" : $question.title,
  "answer score" : $answer.score
}
```

Results:

```
{
  "question" : "MySQL and NoSQL: Help me to choose the right
 one",
  "answer score" : 17
}
{
  "question" : "MySQL and NoSQL: Help me to choose the right
 one",
  "answer score" : 1
}
```

```
{
  "question" : "Redis, CouchDB or Cassandra?",
  "answer score" : 34
}
{
  "question" : "Full-text search in NoSQL databases",
  "answer score" : 6
}
{
  "question" : "Find CouchDB docs missing an arbitrary field
",
  "answer score" : 0
}
{
  "question" : "Find CouchDB docs missing an arbitrary field
",
  "answer score" : 1
}
```

Order Clauses

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 a.display_name, a.score FROM answers a ORDER BY a.display_name".

Example 10.10. An order by clause.

```
for $answer in collection("answers")
order by $answer.owner.display_name
return {
  "owner" : $answer.owner.display_name,
  "score" : $answer.score
}
```

Results:

```
{
  "owner" : "JasonSmith",
  "score" : 34
}
{
  "owner" : "JasonSmith",
  "score" : 6
}
{
  "owner" : "JasonSmith",
  "score" : 0
}
{
  "owner" : "JasonSmith",
  "score" : 1
}
{
  "owner" : "Rob Wells",
  "score" : 4
}
{
  "owner" : "Ubiguchi",
  "score" : 7
}
{
  "owner" : "Victor Nicollet",
  "score" : 17
}
{
  "owner" : "descent89",
  "score" : 1
}
```

Multiple sorting criteria can be given - they are treated with the semantics of a lexicographic order, that is, incoming tuples are first sorted according to the first criterion, and in case of equality the second criterion is used, etc.

Example 10.11. An order by clause with two criteria.

```
for $answer in collection("answers")
order by $answer.owner.display_name,
         $answer.score
return {
  "owner" : $answer.owner.display_name,
  "score" : $answer.score
}
```

Results:

```
{
  "owner" : "JasonSmith",
  "score" : 0
}
{
  "owner" : "JasonSmith",
  "score" : 1
}
{
  "owner" : "JasonSmith",
  "score" : 6
}
{
  "owner" : "JasonSmith",
  "score" : 34
}
{
  "owner" : "Rob Wells",
  "score" : 4
}
{
  "owner" : "Ubiguchi",
  "score" : 7
}
{
  "owner" : "Victor Nicollet",
  "score" : 17
}
{
  "owner" : "descent89",
  "score" : 1
}
```

For each criterion, 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 (even before null) or last (even after null).

Example 10.12. An order by clause with ordering options.

```
for $answer in collection("answers")
order by $answer.owner.display_name
             descending empty greatest,
         $answer.score ascending
return {
  "owner" : $answer.owner.display_name,
  "score" : $answer.score
}
```

Results:

```
{
  "owner" : "descent89",
  "score" : 1
}
{
  "owner" : "Victor Nicollet",
  "score" : 17
}
{
  "owner" : "Ubiguchi",
  "score" : 7
}
{
  "owner" : "Rob Wells",
  "score" : 4
}
{
  "owner" : "JasonSmith",
  "score" : 0
}
{
  "owner" : "JasonSmith",
  "score" : 1
}
{
  "owner" : "JasonSmith",
  "score" : 6
}
{
  "owner" : "JasonSmith",
  "score" : 34
}
```

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

Group Clauses

Grouping is also supported, like in SQL.

For each incoming tuple, the expression in the group clause is evaluated to an atomic. The value of this atomic is called a grouping key. The incoming tuples are then grouped according to the grouping key -- one group for each value of the grouping key.

For each group, a tuple is output, in which:

- Each grouping variable (appearing in the group clause) is bound to the group's key corresponding to this variable.
- Each other (non-grouping) variable is bound to the sequence obtained by concatenating all original values of the variable within the group. Aggregations can then be done on these variables in further clauses.

Here is an example:

```
for $i in (1, 2),
    $j in (3, 4)
group by $j
...
```

The first for clause produces four tuples (this is again an ad-hoc syntax for illustrative purposes):

```
"$i" : 1, "$j" : 3
"$i" : 1, "$j" : 4
"$i" : 2, "$j" : 3
"$i" : 2, "$j" : 4
```

Then the group clause groups according the value of $j. There are two distinct values (3 and 4), so that this results in two groups.

```
Group 1 (key $j : 3)
$i : 1, $j : 3
$i : 2, $j : 3

Group 2 (key $j : 4)
$i : 1, $j : 4
$i : 2, $j : 4
```

In each output tuple, $j is the grouping variable and is bound to the key of the group. $i is non-grouping and is bound to the sequence of all values in the group.

```
$i : (1, 2), $j : 3
$i : (1, 2), $j : 4
```

The following query is equivalent to "SELECT question_id FROM answers GROUP BY question_id".

Example 10.13. A group by clause.

```
for $answer in collection("answers")
group by $question := $answer.question_id
return { "question" : $question }
```

Results:

```
{
  "question" : 5453872
}
{
  "question" : 6183352
}
{
  "question" : 4720508
}
{
  "question" : 4419499
}
{
  "question" : 37823
}
```

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

Example 10.14. A group by clause using count aggregation.

```
for $answer in collection("answers")
group by $question := $answer.question_id
return {
  "question" : $question,
  "count" : count($answer)
}
```

Results:

```
{
  "question" : 5453872,
  "count" : 1
}
{
  "question" : 6183352,
  "count" : 2
}
{
  "question" : 4720508,
  "count" : 1
}
{
  "question" : 4419499,
  "count" : 2
}
```

```
{
  "question" : 37823,
  "count" : 2
}
```

The following query is equivalent to "SELECT question_id, AVG(score) FROM answers GROUP BY question_id".

Example 10.15. A group by clause using average aggregation.

```
for $answer in collection("answers")
group by $question := $answer.question_id
return {
  "question" : $question,
  "average score" : avg($answer.score)
}
```

Results:

```
{
  "question" : 5453872,
  "average score" : 6
}
{
  "question" : 6183352,
  "average score" : 0.5
}
{
  "question" : 4720508,
  "average score" : 34
}
{
  "question" : 4419499,
  "average score" : 9
}
{
  "question" : 37823,
  "average score" : 5.5
}
```

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

Example 10.16. A group by clause with a nested expression.

```
for $answer in collection("answers")
group by $question := $answer.question_id
return {
  "question" : $question,
  "scores" : [ $answer.score ]
}
```

Results:

```
{
  "question" : 5453872,
  "scores" : [ 6 ]
}
{
  "question" : 6183352,
  "scores" : [ 0, 1 ]
}
{
  "question" : 4720508,
  "scores" : [ 34 ]
}
{
  "question" : 4419499,
  "scores" : [ 17, 1 ]
}
{
  "question" : 37823,
  "scores" : [ 7, 4 ]
}
```

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 question_id, COUNT(*) FROM answers GROUP BY question_id HAVING COUNT(*) > 1"

Example 10.17. A group by clause with a post-grouping condition.

```
for $answer in collection("answers")
group by $question := $answer.question_id
where count($answer) gt 1
return {
  "question" : $question,
  "count" : count($answer)
}
```

Results:

```
{
  "question" : 6183352,
  "count" : 2
}
{
  "question" : 4419499,
  "count" : 2
}
{
  "question" : 37823,
  "count" : 2
}
```

Let Clauses

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 10.18. A let clause.

```
for $answer in collection("answers")
let $qid := $answer.question_id
group by $question := $qid
let $count := count($answer)
where $count gt 1
return {
  "question" : $question,
  "count" : $count
}
```

Results:

```
{
  "question" : 6183352,
  "count" : 2
}
{
  "question" : 4419499,
  "count" : 2
}
{
  "question" : 37823,
  "count" : 2
}
```

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

Example 10.19. A let clause reusing the same variable name.

```
for $answer in collection("answers")
let $qid := $answer.question_id
group by $qid
let $count := count($answer)
where $count gt 1
let $count := sum(
  collection("faqs")
    [ $$.question_id eq $qid ]!size($$.tags)
)
return {
  "question" : collection("faqs")
      [$$.question_id eq $qid].title,
  "count" : $count
}
```

Results:

```
{
  "question" : "Find CouchDB docs missing an arbitrary field
",
  "count" : 4
}
{
  "question" : "MySQL and NoSQL: Help me to choose the right
 one",
  "count" : 4
}
{
  "question" : null,
  "count" : 0
}
```

Count Clauses

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 10.20. A count clause.

```
for $question in collection("faqs")
order by size($question.tags)
count $count
return {
  "id" : $count,
  "faq" : $question.title
}
```

Results:

```
{
  "id" : 1,
  "faq" : "MySQL and NoSQL: Help me to choose the right one"

}
{
  "id" : 2,
  "faq" : "The Next-gen Databases"
}
```

```
{
  "id" : 3,
  "faq" : "Redis, CouchDB or Cassandra?"
}
{
  "id" : 4,
  "faq" : "Find CouchDB docs missing an arbitrary field"
}
{
  "id" : 5,
  "faq" : "Full-text search in NoSQL databases"
}
```

Map Operator

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

Example 10.21. A simple map.

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

Results:

```
2
4
6
8
10
12
14
16
18
20
```

Example 10.22. An equivalent query.

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

Results:

```
2
4
6
8
10
12
14
16
18
20
```

Composing FLWOR Expressions

Like all other expressions, FLWOR expressions can be composed. In the following example, a predicate expression is nested in an existential quantifier, nested in the where clause of a FLWOR, nested in a function call, nested in a FLWOR, nested in a function call, nested in an array constructor. The examples looks for users who got an answer not accepted, but for whom there were at least two questions for which they gave an answer with a better score.

Example 10.23. Nested FLWORs.

```
[
  distinct-values(
    for $answer in collection("answers")
    let $oid := $answer.owner.user_id
    where count(
      for $question in collection("faqs")
      where
        some $other-answer
        in collection("answers")
           [$$.question_id eq
                $question.question_id
            and
            $$.owner.user_id eq $oid]
        satisfies
          $other-answer.score gt $answer.score
       return $question
    ) ge 2
    where not $answer.is_accepted
    return $answer.owner.display_name
  )
]
```

Results:

```
[ "JasonSmith" ]
```

Ordered and Unordered Expressions

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 10.24. An unordered expression.

```
unordered {
  for $answer in collection("answers")
  where $answer.score ge 4
  count $c
  where $c le 2
  return $answer
}
```

Results:

```
{
  "_id" : "511C7C5D9A277C22D13880C3",
  "question_id" : 37823,
  "answer_id" : 37841,
  "creation_date" : "2008-09-01T12:14:38",
  "last_activity_date" : "2008-09-01T12:14:38",
  "score" : 7,
  "is_accepted" : false,
  "owner" : {
    "user_id" : 2562,
    "display_name" : "Ubiguchi",
    "reputation" : 1871,
    "user_type" : "registered",
    "profile_image" : "http://www.gravatar.com/avatar/00b87a
917ec763c0c051dc6b8c06f402?d=identicon&amp;r=PG",
    "link" : "http://stackoverflow.com/users/2562/ubiguchi"
  }
}
{
  "_id" : "511C7C5D9A277C22D13880C4",
  "question_id" : 37823,
  "answer_id" : 37844,
  "creation_date" : "2008-09-01T12:16:40",
  "last_activity_date" : "2008-09-01T12:16:40",
  "score" : 4,
  "is_accepted" : false,
  "owner" : {
    "user_id" : 2974,
    "display_name" : "Rob Wells",
    "reputation" : 17543,
    "user_type" : "registered",
    "profile_image" : "http://www.gravatar.com/avatar/876928
1d99f8fe9c208fd6a926c383d1?d=identicon&amp;r=PG",
    "link" : "http://stackoverflow.com/users/2974/rob-wells"
,
    "accept_rate" : 94
  }
}
```

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

Example 10.25. An ordered expression.

```
unordered {
  for $question in collection("faqs")
  where exists(
    ordered {
      for $answer at $i in collection("answers")
      where $i eq 5
      where $answer.question_id
          eq $question.question_id
      return $answer
    }
  )
  return $question
}
```

Results:

```
{
  "_id" : "511C7C5C9A277C22D138808A",
  "question_id" : 4720508,
  "creation_date" : "2011-01-18T04:32:30",
  "last_activity_date" : "2011-01-19T06:46:34",
  "score" : 13,
  "accepted_answer_id" : 4720977,
  "title" : "Redis, CouchDB or Cassandra?",
  "tags" : [ "nosql", "couchdb", "cassandra", "redis" ],
  "view_count" : 5620,
  "owner" : {
    "user_id" : 216728,
    "display_name" : "nornagon",
    "reputation" : 3114,
    "user_type" : "registered",
    "profile_image" : "http://www.gravatar.com/avatar/13f271
99f9bf9c9f1261dc8a49630a6b?d=identicon&amp;r=PG",
    "link" : "http://stackoverflow.com/users/216728/nornagon
",
    "accept_rate" : 86
  },
  "link" : "http://stackoverflow.com/questions/4720508/redis
-couchdb-or-cassandra",
  "is_answered" : true
}
```

Chapter 11. Expressions Dealing with Types

We have already introduced the sequence type syntax. It is now time to introduce the expressions that deal with types.

Instance-of Expressions

A quick glimpse on this expression was already given. An instance expression can be used to tell whether a sequence matches a given sequence type, like in Java.

Example 11.1. Instance of expression.

```
1 instance of integer,
1 instance of string,
"foo" instance of string,
{ "foo" : "bar" } instance of object,
({ "foo" : "bar" }, { "bar" : "foo" })
    instance of json-item+,
[ 1, 2, 3 ] instance of array?,
() instance of ()
```

Results:

```
true
false
true
true
true
true
true
```

Treat Expressions

A treat expression just forwards its operand value, but only after checking that a JSONiq value matches a given sequence type. If it is not the case, an error is raised.

Example 11.2. Treat as expression.

```
1 treat as integer,
"foo" treat as string,
{ "foo" : "bar" } treat as object,
({ "foo" : "bar" }, { "bar" : "foo" })
    treat as json-item+,
[ 1, 2, 3 ] treat as array?,
() treat as ()
```

Results:

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

Example 11.3. Treat as expression (failing).

```
1 treat as string
```

Error:

```
"xs:integer" cannot be treated as type xs:string
```

Castable Expressions

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.

The question mark allows for an empty sequence.

Example 11.4. Castable as expression.

```
"1" castable as integer,
"foo" castable as integer,
"2013-04-02" castable as date,
() castable as date,
("2013-04-02", "2013-04-03") castable as date,
() castable as date?
```

Results:

```
true
false
true
false
false
true
```

Cast Expressions

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

Also here, the question mark allows for an empty sequence. An error is raised if the cast is unsuccessful.

Example 11.5. Cast as expression.

```
"1" cast as integer,
"2013-04-02" cast as date,
() cast as date?,
"2013-04-02" cast as date?
```

Results:

```
1
```

```
"2013-04-02"
"2013-04-02"
```

Example 11.6. Cast as expression (failing).

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

Error:

```
sequence of more than one item can not be cast to type with
 quantifier '1' or '?'
```

Typeswitch Expressions

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 11.7. Typeswitch expression.

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

Results:

```
"string"
```

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

Example 11.8. 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
```

Results:

```
"foofoo"
```

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

Example 11.9. Typeswitch expression.

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

Results:

```
{
  "integer or string" : "foo"
}
```

Part IV. Prolog, Modules and Functions

Chapter 12. Prologs

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

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.

Setters.

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

Default Ordering Mode

This specifies the default behaviour of for 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.

Example 12.1. A default ordering setter.

```
declare ordering unordered;
for $answer in collection("answers")
return {
  "owner" : $answer.owner.display_name,
  "score" : $answer.score
}
```

Results:

```
{
  "owner" : "Ubiguchi",
  "score" : 7
}
{
  "owner" : "Rob Wells",
  "score" : 4
}
```

```
{
  "owner" : "Victor Nicollet",
  "score" : 17
}
{
  "owner" : "descent89",
  "score" : 1
}
{
  "owner" : "JasonSmith",
  "score" : 34
}
{
  "owner" : "JasonSmith",
  "score" : 6
}
{
  "owner" : "JasonSmith",
  "score" : 0
}
{
  "owner" : "JasonSmith",
  "score" : 1
}
```

Default Ordering Behaviour for Empty Sequences

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

Example 12.2. A default ordering for empty sequences.

```
declare default order empty least;
for $x in ({ "foo" : "bar" }, {})
order by $x.foo
return $x
```

Results:

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

Default Decimal Format

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

Example 12.3. A default decimal format setter.

```
declare default decimal-format
    decimal-separator = ","
    grouping-separator = " ";
format-number(12345.67890, "# ###,##")
```

Results:

```
"12 345,68"
```

Namespaces

Variables and functions live in namespaces that are URIs -- the semantics is similar to that of C++ namespaces. For convenience, namespaces are associated with a much shorter alias, and this alias can be used as a prefix to a variable or a function.

Until now, we only dealt with main queries. In main queries, the namespace alias *local:* is predefined so that global variables and functions that are local to the main query can use this alias, for example *local:myvariable* or *local:myfunction()*. This alias is associated with a namespace, but which namespace it is not relevant for writing queries.

For variables, the alias is optional -- variables not prefixed with an alias live in no namespace.

For functions, the absence of alias is only allowed for builtin functions. Builtin functions live in their own special namespace.

Other namespaces and aliases can be defined as well with imported library modules. This is defined in Chapter 13, *Modules*.

Global Variables

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

Example 12.4. Global variable.

```
declare variable $obj
  := { "foo" : "bar" };
declare variable $numbers
    := (1, 2, 3, 4, 5);
$obj,
[ $numbers ]
```

Results:

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

You can specify a sequence type for a variable. If the type does not match, an error is raised. 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 12.5. Global variable with a type.

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

Results:

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

An external variable allows you to pass a value from the outside environment, which can be very useful. Each implementation can choose its 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 from outside.

Example 12.6. An external global variable with a default value.

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

Results:

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

In these examples, global variables have no prefix. They can also be prefixed with the predefined alias *local:*, but them they must be prefixed both in the declaration and when used.

Example 12.7. An external global variable with the local: alias.

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

Results:

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

Global variables that are imported from other modules are prefixed with the alias associated with the imported module, as will be explained in Chapter 13, *Modules*.

User-Defined Functions

You can define your own functions in the prolog.

Unlike variables, user-defined functions must be prefixed, because unprefixed functions are the builtin functions.

In the prolog of a main query, these user-defined functions must be prefixed with the predefined alias *local:*, both in the declaration and when called.

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

Example 12.8. Some user-defined functions.

```
declare function local:say-hello-1($x)
{
  "Hello, " || $x || "!"
};

declare function local:say-hello-2($x as string)
{
  "Hello, " || $x || "!"
};

declare function local:say-hello-3($x as string)
    as string
{
  "Hello, " || $x || "!"
};

local:say-hello-1("Mister Spock"),
local:say-hello-2("Mister Spock"),
local:say-hello-3("Mister Spock")
```

Results:

```
"Hello, Mister Spock!"
"Hello, Mister Spock!"
"Hello, Mister Spock!"
```

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

Example 12.9. A type mismatch for a user-defined function (failing).

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

local:say-hello(1)
```

Error:

```
xs:integer can not be promoted to parameter type xs:string
of function local:say-hello()
```

Chapter 13. Modules

You can group functions and variables in separate units, called library modules.

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

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 13.1. 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
};
```

Once you have defined a library module, you can import it in any other module (library or main). An alias must be 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 -- only the namespace really matters.

Example 13.2. Importing a library module into a main module.

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

Results:

```
1764
```

An engine may come with a number of builtin library modules. For example, there is the standardized math module.

Example 13.3. Using the math module.

```
import module namespace math =
    "http://www.w3.org/2005/xpath-functions/math";
math:pi(),
math:pow(2, 30),
math:exp(1),
math:exp10(2),
math:log(1),
math:log10(2),
math:sqrt(4),
math:sin(math:pi())
```

Results:

```
3.141592653589793l
1.073741824E9
2.718281828459045l
100
0
0.301029995663981 2
2
1.2246467991473532E-16
```

Chapter 14. Function Library

JSONiq provides a rich set of builtin functions. We now introduce them, mostly by giving examples of usage.

Example 14.1. Functions on JSON data.

```
keys({ "foo" : 1, "bar" : 2 }),
members([ "mercury", "venus", "earth", "mars" ]),
parse-json(
"{ \"foo\" : \"bar\" }"
),
size([1 to 10]),
serialize({ "foo" : "bar" })
```

Results:

```
"foo"
"bar"
"mercury"
"venus"
"earth"
"mars"
{
  "foo" : "bar"
}
10
"{ "foo" : "bar" }"
```

Example 14.2. Miscellaneous functions.

```
collection("one-object"),
boolean("foo"),
if (1 + 1 ne 2) then error() else true
```

Results:

```
{
  "question" : "What NoSQL technology should I use?"
}
true
true
```

Example 14.3. Functions on numbers.

```
abs(-2.3),
ceiling(-2.3),
floor(-2.3),
round(-2.3),
round-half-to-even(-2.5145, 3),
number("3.14"),
format-integer(1234567, "000'111'222'333"),
format-number(1234567.8901234, "#,###.123")
```

Results:

```
2.3
-2
-3
-2
-2.514
3.14
"000'001'234'567"
"1,234,567.890"
```

Example 14.4. Functions on strings (1/2).

```
codepoints-to-string((78, 111, 83, 81, 76)),
string-to-codepoints("NoSQL"),
codepoint-equal(
    "NoSQL",
    "\u004E\u006F\u0053\u0051\u004C"
),
upper-case("NoSQL"),
lower-case("NoSQL"),
translate("NoSQL", "oN", "On"),
resolve-uri("types", "http://www.jsoniq.org/"),
encode-for-uri("1 + 1 is 2"),
iri-to-uri(
    "http://www.example.com/chuchichäschtli"),
escape-html-uri(
    "http://www.example.com/chuchichäschtli")
```

Results:

```
"NoSQL"
78
111
83
81
76
true
"NOSQL"
"nosql"
"nOSQL"
"http://www.jsoniq.org/types"
"1%20%2B%201%20is%202"
"http://www.example.com/chuchich%C3%A4schtli"
"http://www.example.com/chuchich%C3%A4schtli"
```

Example 14.5. Functions on strings (2/2).

```
concat("foo", 1, true, "bar", ()),
string-join((1 to 10) ! string($$), "-"),
string-length("123456789"),
contains("NoSQL", "SQL"),
starts-with("NoSQL", "No"),
ends-with("NoSQL", "SQL"),
substring("123456789", 5),
substring-before("NoSQL", "SQL"),
substring-after("NoSQL", "o"),
matches("NoSQL", "No[A-Z]+"),
replace("NoSQL", "No([A-Z])", "Yes$1"),
tokenize(
    "Go Boldly Where No Man Has Gone Before",
    " "
)
```

Results:

```
"foo1truebar"
"1-2-3-4-5-6-7-8-9-10"
9
true
true
true
"56789"
"No"
"SQL"
true
"YesSQL"
"Go"
"Boldly"
"Where"
"No"
"Man"
"Has"
"Gone"
"Before"
```

Example 14.6. Functions on sequences (1/2).

```
empty(("foo", "bar")),
exists(("foo", "bar")),
head(("foo", "bar")),
tail(("foo", "bar")),
[ insert-before(("foo", "bar"), 1, "foobar") ],
remove(("foo", "bar"), 1),
[ reverse(1 to 10) ],
[ subsequence(1 to 10, 2, 4) ],
unordered(("foo", "bar"))
```

Results:

```
false
true
"foo"
```

```
"bar"
[ "foobar", "foo", "bar" ]
"bar"
[ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
[ 2, 3, 4, 5 ]
"foo"
"bar"
```

Example 14.7. Functions on sequences (2/2).

```
distinct-values(
  ("foo", "bar", "foo", "bar", "foo")
),
index-of(
  ("foo", "bar", "foo", "bar", "foo"),
  "foo"),
deep-equal(
  { "foo" : [ 1 to 10 ] },
  { lower-case("FOO") : [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ] }
),
zero-or-one("foo"),
one-or-more(("foo", "bar")),
exactly-one("foo")
```

Results:

```
"foo"
"bar"
1
3
5
true
"foo"
"foo"
"bar"
"foo"
```

Example 14.8. Aggregation functions.

```
count(1 to 100),
avg(1 to 100),
max(1 to 100),
min(1 to 100),
sum(1 to 100)
```

Results:

```
100
50.5
100
1
5050
```

Example 14.9. Environment.

```
current-dateTime(),
current-date(),
current-time(),
implicit-timezone()
```

Results:

```
"2013-06-10T15:23:24.163953+02:00"
"2013-06-10+02:00"
"15:23:24.163953+02:00"
"PT2H"
```

Example 14.10. Constructors.

```
date("2013-06-21"),
time("17:00:00"),
dateTime("2013-06-21T17:00:00Z"),
dateTime("2013-06-21T17:00:00+01:00"),
duration("P2DT1H30M15S"),
hexBinary("511C7C5C9A277C22D138802F")
```

Results:

```
"2013-06-21"
"17:00:00"
"2013-06-21T17:00:00Z"
"2013-06-21T17:00:00+01:00"
"P2DT1H30M15S"
"511C7C5C9A277C22D138802F"
```

Part V. Advanced Notes

Chapter 15. Errors

Builtin expressions, operators and functions may raise errors under various conditions. An example is a mismatching type.

The evaluation of a JSONiq expression either returns a sequence of items, or raises an error.

Errors can be reported statically, or dynamically (at runtime).

Errors can also be raised by hand.

Example 15.1. Raising an error (failing).

```
      error()
```

Error:

Lazy evaluation and optimizations with regard to errors are allowed. Raising errors is not always deterministic, as in some cases the processor might (but is not required to) stop evaluating the operands of an expression if it determines that only one possible value can be returned by that expression. The following expression may return true, or may raise an error.

Example 15.2. Non-deterministic behavior (no guarantee of failure or success).

```
      true or error()
```

Results:

```
true
```

Chapter 16. Equality vs. Identity

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

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

Example 16.1. Logical comparison of two atomics.

```
1 eq 1
```

Results:

```
true
```

Example 16.2. Logical comparison of two atomics.

```
1 eq 2
```

Results:

```
false
```

Example 16.3. Logical comparison of two atomics.

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

Results:

```
false
```

Example 16.4. Logical comparison of two atomics.

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

Results:

```
true
```

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

Example 16.5. Logical comparison of two JSON items.

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

Results:

```
true
```

Example 16.6. Logical comparison of two JSON items.

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

Results:

```
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.

Chapter 17. 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 very useful to stream and optimize -- for example, the runtime of the Zorba engine is iterator-based.

Example 17.1. Flat sequences.

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

Results:

```
1
2
3
4
```

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

Example 17.2. Nesting arrays.

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

Results:

```
[ [ 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 the item itself.

Example 17.3. Singleton sequences.

```
1 + 1
```

Results:

```
2
```

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

Example 17.4. Singleton sequences.

```
[ 1 + 1 ]
```

Results:

```
[ 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 17.5. count() on an array.

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

Results:

```
1
```

Example 17.6. count() on a sequence.

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

Results:

```
4
```

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

Example 17.7. Members of an array.

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

Results:

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

Example 17.8. Members of a sequence.

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

Results:

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

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

Example 17.9. Converting an array to a sequence.

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

Results:

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

Example 17.10. Converting a sequence to an array.

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

Results:

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

Chapter 18. 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. Empty sequences cannot, as they represent the absence of any item.

Example 18.1. Null values in an array

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

Results:

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

Example 18.2. Null values in an object

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

Results:

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

Example 18.3. Null values in a sequence

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

Results:

```
null
1
```

```
null
2
```

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

Example 18.4. Automatic conversion to null.

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

Results:

```
{
  "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 18.5. Empty sequence in an arithmetic operation.

```
() + 2
```

Results:

Example 18.6. Null in an arithmetic operation (failing).

```
null + 2
```

Error:

```
arithmetic operation not defined between types "js:null" an
d "xs:integer"
```

Example 18.7. Null and empty sequence in an arithmetic operation.

```
null + ()
```

Results:

Example 18.8. Empty sequence in a comparison.

```
() eq 2
```

Results:

Example 18.9. Null in a comparison.

```
null eq 2
```

Results:

```
false
```

Example 18.10. Null in a comparison.

```
null lt 2
```

Results:

```
true
```

Example 18.11. Null and the empty sequence in a comparison.

```
null eq ()
```

Results:

Example 18.12. Null and the empty sequence in a comparison.

```
null lt ()
```

Results:

Chapter 19. Reference

A great part of JSONiq is directly inherited from XQuery -- everything that is orthogonal to XML.

If you would like to know more about JSONiq, you can browse *http://www.jsoniq.org/*.

If you are interested in knowing the semantics of the expressions more in depth, you can find most of them on the XQuery 3.0 specification at *http://www.w3.org/TR/xquery-30*.

If you are interested in knowing the semantics of the builtin functions more in depth, you can find most of them on the XPath and XQuery Functions and Operators 3.0 specification at *http://www.w3.org/TR/xpath-functions-30*.