Simple List Conventions for RDF Version 0.1

This version:
http://goo.gl/8PNuAG
Latest version:
@@TODO@@
Previous version:
@@TODO@@

Authors:
David Booth <david@dbooth.org>

Abstract

The standard rdf:List vocabulary makes it easy to write an ordered list in Turtle or SPARQL using parentheses: (. . .). Unfortunately it is hard to write a SPARQL 1.1 query that will return the items of an rdf:List of unknown length *in their original order.* Hopefully the RDF and SPARQL standards will be enhanced someday to provide better support for lists, to address this problem. As an interim solution, this document defines simple conventions and a tiny vocabulary for representing ordered lists -- arrays actually -- in RDF in a way that is convenient for standard SPARQL queries. Explicit indexes are used for element ordering: each element in a list is represented by a *(item index)* pair. This allows standard SPARQL queries to order results by *index*. It also allows specific items -- first, last or any other -- to be looked up directly by *index*, rather than traversing the linked list structure of an rdf:List.

Table of Contents

Status of This Document

This is a draft, with no official status. Please add comments or suggestions directly to this document using *Insert->Comment*.

Namespace

The vocabulary defined herein uses the following namespace:

```
PREFIX list: <http://purl.org/NET/list#>
```

Quick Start

There are two ways to use these conventions. The short form is good when you have an RDF property that is intended to hold exactly one list, in which case each value of that property can represent one element in the list. Notice that no prefix declaration is needed when a Simple List is used this way, though you may choose to include one to make your data more self-describing, as discussed under Property list:item.

SHORT Form -- Write this Turtle	Instead of this

`PREFIX : <http://example/>`
`:jane :likes`
`( "bananas" 0 ) ,`
`( "apples" 1 ) ,`
`( "oranges" 2 ) .`
	`PREFIX : <http://example/>`
	`:jane :likes (`
	`"bananas"`
	`"apples"`
	`"oranges"`
	`) .`

Query like this	Result

`# List all items in order:`
`PREFIX : <http://example/>`
`SELECT ?item {`
`:jane :likes ( ?item ?index ) .`
`} ORDER BY ?index`
	?item
	"bananas"
	"apples"
	"oranges"

The long form is needed when your RDF property may hold multiple separate lists. It uses the list:item property to specify list elements; hence the additional prefix declaration.

LONG Form -- Write this Turtle	Instead of this

`PREFIX list: <http://purl.org/NET/list#>`
`PREFIX : <http://example/>`
`:john :loves :petList .`
`:petList list:item`
`( "cats" 0 ) ,`
`( "dogs" 1 ) ,`
`( "fish" 2 ) .`
	`PREFIX : <http://example/>`
	`:john :loves :petList .`
	`:petList :list (`
	`"cats"`
	`"dogs"`
	`"fish"`

```
:john :loves :peopleList .
:peopleList list:item
      ( "jane" 0 ) ,
      ( "sarah" 1 ) ,
      ( "paul" 2 ) .
```

```
      ) .
:john :loves :peopleList .
:peopleList :list (
      "jane"
      "sarah"
      "paul"
      ) .
```

Query like this

```
# Get all lists and items in order:
PREFIX list: <http://purl.org/NET/list#>
PREFIX : <http://example/>
SELECT ?list ?item {
  :john :loves ?list .
  ?list list:item ( ?item ?index ) .
} ORDER BY ?list ?index
```

Result

?list	?item
:peopleList	"jane"
:peopleList	"sarah"
:peopleList	"paul"
:petList	"cats"
:petList	"dogs"
:petList	"fish"

That's all you will usually need, but there is also a list:List class and a list:length property if you want to use them.

Classes and Properties

Namespaces referenced below:

```
PREFIX list: <http://purl.org/NET/list#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
```

Class list:List

```
list:List a rdfs:Class , owl:Class .
```

Optional. Indicates that the object follows the ordered list conventions defined herein. A list:List object may have zero or more list:item elements, each of which MUST be an rdf:List of exactly two elements: *(item index)*. Items are numbered consecutively starting at 0. More precisely,

each *index* MUST be unique within that list:List and MUST be a non-negative integer in the inclusive range [0, *n*-1], where *n* is the number of elements associated with the list:List object under the closed world assumption, i.e., the number of *(item index)* pairs that have been asserted. Example:

```
:petList a list:List .
```

Property list:item

```
list:item a rdf:Property , owl:ObjectProperty .
  rdfs:domain list:List ;
  rdfs:range rdf:List .  # Of exactly two items: (item index)
```

Specifies one element in a list:List -- an *(item index)* pair. The *item* is first in the pair because that makes it closer to access when doing a join against other triples, and when the *index* is not needed in the query. (See the Get all items UNORDERED -- method 2 (faster) example.) Examples:

```
:petList list:item ( "cats" 0 ) .
:petList list:item ( "dogs" 1 ) .
:petList list:item ( "fish" 2 ) .
```

Often an application-specific property is used instead of the list:item property, such as:

```
:jane :likes ( "bananas" 0 ) .
:jane :likes ( "apples"  1 ) .
:jane :likes ( "oranges" 2 ) .
```

In this case, the application-specific property :likes is used as a subproperty of list:item, and MAY be declared:

```
:likes rdfs:subPropertyOf list:item .
```

Explicitly declaring :likes to be a subproperty of list:item makes the data more self-describing. However, if you know that :likes holds list:items then you can still query for them without an explicit subproperty assertion.

Property list:length

```
list:length a rdf:Property , owl:DatatypeProperty ;
  rdfs:domain list:List ;
  rdfs:range xsd:nonNegativeInteger .
```

Optional. Indicates the number of items in the list. It is not automatically set: you must set it if you want to use it. Example:

```
:petList a list:List ;
```

```
            list:length 3 ;
            list:item ( "cats" 0 ) ,
                      ( "dogs" 1 ) ,
                      ( "fish" 2 ) .
```

More Query Examples

Examples in this section are based on the following data:

```
PREFIX : <http://example/>
:jane :likes
      ( "bananas" 0 ) ,
      ( "apples"  1 ) ,
      ( "oranges" 2 ) .
```

Count the items

SPARQL Query	Result

`PREFIX : <http://example/>`
`SELECT (COUNT(?pair) as ?count) {`
`  :jane :likes ?pair .`
`}`
	?count
	3

Get the first item (zero-based index)

SPARQL Query	Result

`PREFIX : <http://example/>`
`SELECT ?item {`
`  :jane :likes ( ?item 0 ) .`
`}`
	?item
	"bananas"

Get the second item (zero-based index)

Any item in the list can be accessed directly using its index. This example gets the second item in the list:List.

SPARQL Query	Result

`PREFIX : <http://example/>`
`SELECT ?item {`
`  :jane :likes ( ?item 1 ) .`	

```
}
```

?item
"apples"

Get the last item (using LIMIT)

SPARQL Query	Result

```
PREFIX : <http://example/>
SELECT ?item {
  :jane :likes ( ?item ?index ) .
} ORDER BY DESC(?index)
LIMIT 1
```

?item
"oranges"

Get the last item (using a subquery)

SPARQL Query	Result

```
PREFIX : <http://example/>
SELECT ?item {
  { SELECT (COUNT(?pair)-1 as ?lastIndex)
      { :jane :likes ?pair . } }
  :jane :likes ( ?item ?lastIndex ) .
}
```

?item
"oranges"

Get the last item (if list:length is set)

SPARQL Query	Result

```
PREFIX list: <http://purl.org/NET/list#>
PREFIX : <http://example/>
SELECT ?item {
  :jane list:length ?length .
  BIND( ?length-1 as ?lastIndex ) .
  :jane :likes ( ?item ?lastIndex ) .
}
```

?item
"oranges"

Count the items and assert list:length

SPARQL UPDATE operation

```
PREFIX list: <http://purl.org/NET/list#>
PREFIX : <http://example/>
INSERT {
```

```
  :jane list:length ?length .
}
WHERE {
  SELECT (COUNT(?pair) as ?length)
      { :jane :likes ?pair . }
}
```

Get the last two items

SPARQL Query	Result

`PREFIX : <http://example/>`
`SELECT ?item {`
`  :jane :likes ( ?item ?index ) .`
`} ORDER BY DESC(?index)`
`LIMIT 2`
	?item
	"oranges"
	"apples"

Get all items UNORDERED -- method 1 (slower)

SPARQL Query	Result

`PREFIX : <http://example/>`
`SELECT ?item {`
`  :jane :likes ( ?item ?index ) .`
`}`
	?item
	"oranges"
	"bananas"
	"apples"
	(or any other order)

Get all items UNORDERED -- method 2 (faster)

For one million items on a 2011 quad core laptop using Sesame, this method took 2 seconds, whereas method 1 took 15 seconds. You may not be using million-item lists, but you probably will be joining against items in your lists, and joins tend to multiply the execution time required, so the difference in speed between methods 1 and 2 could still be relevant on small lists.

SPARQL Query	Result

`PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
`PREFIX : <http://example/>`
`SELECT ?item {`
	?item

```
  :jane :likes / rdf:first ?item .
}
```

"oranges"
"bananas"
"apples"

(or any other order)

Convert an rdf:List to a Simple List Conventions list:List

This example reads an rdf:List associated with :jane and creates a new list:List associated with :newJane using the Simple List Conventions. WARNING: SLOW! It would be much more efficient to use the Jena ARQ al:index magic property to do this, or a non-SPARQL technique. See the example of Jena ARQ extension for list item indexes.

SPARQL 1.. UPDATE operation

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example/>
INSERT {
  :newJane :likes ( ?item ?index ) .
}
WHERE {
  SELECT ?item (COUNT(?mid)-1 as ?index) {
       :jane :likes / rdf:rest* ?mid .
       ?mid rdf:rest* ?node .
       ?node rdf:first ?item .
  } GROUP BY ?node ?item
}
```

Background: What's the problem?

The standard rdf:List vocabulary makes it easy to write an ordered list in Turtle or SPARQL. But it is hard to write a SPARQL query that will return the items of an rdf:List (of unknown length) *in their original order.*

Problem examples

For example, consider :jane's top three favorite fruits, in ranked order:

Given this list in Turtle	It is hard to write a standard SPARQL query to return this
`PREFIX : <http://example/>` `:jane :likes (`	?item

"bananas" "apples" "oranges") .	"bananas" "apples" "oranges"

The following naive SPARQL query will return all of the items, but it does not guarantee their order:

SPARQL Query	Result

# WRONG! Order is NOT guaranteed! 	
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example/>
SELECT ?item {
:jane :likes ?list .
?list rdf:rest* / rdf:first ?item .
}
	?item
	"oranges"
	"bananas"
	"apples"
	(or any other order)

Simply adding an ORDER BY clause will not help either, because that will merely return the items in alphabetical order -- not in their original list order:

SPARQL Query	Result
# WRONG! Alphabetical order -- NOT list order! PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX : <http://example/> SELECT ?item { :jane :likes ?list . ?list rdf:rest* / rdf:first ?item . } ORDER BY ?item	?item "apples" "bananas" "oranges"

Getting rdf:List items in order using standard SPARQL

It is *possible* to select the items in order using standard SPARQL, but it is not easy or efficient. Here is an example, adapted from a technique by Joshua Taylor:

SPARQL Query	Result

`# Correct, but complex and SLOW!`
`PREFIX rdf:`
`<http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
`PREFIX : <http://example/>`
`SELECT ?item (COUNT(?mid) as ?pos) {`
`  :jane :likes / rdf:rest* ?mid .`
`  ?mid rdf:rest* ?node .`
`  ?node rdf:first ?item .`
`} GROUP BY ?node ?item`
`  ORDER BY ?pos`
	<table><tr><th>?item</th><th>?pos</th></tr><tr><td>"bananas"</td><td>1</td></tr><tr><td>"apples"</td><td>2</td></tr><tr><td>"oranges"</td><td>3</td></tr></table>

Other Proposed Solutions

Standard rdf:List

The baseline for comparison is the standard rdf:List. As shown in the example above, it is *possible* to retain the order of items in a list using standard SPARQL, but it is not easy. On the other hand, standard rdf:Lists are more memory-efficient than using the Simple List Conventions: an rdf:List uses two triples per item; the Simple List Conventions use five triples per item.

Enhancing the RDF and/or SPARQL standards

At the 2009 W3C RDF Next Steps Workshop David Wood and James Leigh proposed an enhancement to RDF that would provide better support for lists. This would be the ideal solution, but the proposal was not included in the working group charter when the RDF standard was updated to version 1.1 in 2011-2014, and as of this writing there are no plans on the horizon for a new W3C RDF working group.

Jena ARQ extension for list item indexes

Jena's ARQ library has a function that provides an easy solution, but it is a non-standard SPARQL extension. It uses a "magic property" to expose the index of each item in an rdf:LIst. This is an excellent solution if you do not mind using vendor-specific SPARQL. Here is an example adapted from one by Jose Emilio Labra Gayo:

SPARQL Query	Result

`# WARNING: NON-STANDARD!  Requires Jena/ARQ!`
`PREFIX :     <http://example.org/>`
`PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
`PREFIX al: <http://jena.hpl.hp.com/ARQ/list#>`
`SELECT ?item {`
` :jane :likes ?list .`
	<table><tr><th>?item</th></tr><tr><td>"bananas"</td></tr></table>

?list al:index (?index ?item). } ORDER BY ?index	"apples" "oranges"

Comparison with the Ordered list Ontology (OLO)

The Ordered List Ontology (OLO) is similar to the Simple List Conventions in that it also uses explicit indexes, but it differs in that it uses OLO-specific predicates olo:slot, olo:item, olo:index for accessing item/index pairs. This makes use of OLO ordered lists more verbose. On the other hand, OLO is more memory efficient than the Simple List Conventions, using three triples per item instead of five. Query time appears to be about the same, according to some simple tests on Sesame with a list of one million items. Here is an example of using OLO.

Turtle RDF

Simple List Conventions

```
PREFIX : <http://example/>
:jane :likes
      ( "bananas" 0 ) ,
      ( "apples"  1 ) ,
      ( "oranges" 2 ) .
```

Ordered List Ontology (OLO)

```
PREFIX olo:
<http://purl.org/ontology/olo/core#>
PREFIX : <http://example/>
:jane :likes [
  olo:length 3 ;
  olo:slot
   [ olo:item "bananas" ; olo:index 1 ] ,
   [ olo:item "apples" ;  olo:index 2 ] ,
   [ olo:item "oranges" ; olo:index 3 ]
  ] .
```

SPARQL Query: Get all items in order

Simple List Conventions

```
PREFIX : <http://example/>
SELECT ?item {
  :jane :likes ( ?item ?index ) .
} ORDER BY ?index
```

Ordered List Ontology (OLO)

```
PREFIX olo:
<http://purl.org/ontology/olo/core#>
PREFIX : <http://example/>
SELECT ?item {
  :jane :likes [ olo:slot [
      olo:item ?item ; olo:index ?index ]
  ] .
} ORDER BY ?index
```

Comparison with the Collections Ontology (CO)

The Collections Ontology (CO) is a more OWL-oriented ontology, and also uses explicit item indexes to retain ordering information, using CO-specific predicates co:firstItem, co:index, and co:itemContent . In usage it is very similar to OLO. Here is an example:

<table>
<tr><th colspan="2">Turtle RDF</th></tr>
<tr><th>Simple List Conventions</th><th>Collections Ontology (CO)</th></tr>
<tr><td>

```
PREFIX : <http://example/>
:jane :likes
      ( "bananas" 0 ) ,
      ( "apples"  1 ) ,
      ( "oranges" 2 ) .
```

</td><td>

```
PREFIX co: <http://purl.org/co/List>
PREFIX : <http://example/>
:jane :likes [
  co:size 3 ;
  co:item
   [ co:itemContent "bananas" ; co:index 1 ] ,
   [ co:itemContent "apples" ;  co:index 2 ] ,
   [ co:itemContent "oranges" ; co:index 3 ]
  ] .
```

</td></tr>
<tr><th colspan="2">SPARQL Query: Get all items in order</th></tr>
<tr><th>Simple List Conventions</th><th>Collections Ontology (CO)</th></tr>
<tr><td>

```
PREFIX : <http://example/>
SELECT ?item {
  :jane :likes ( ?item
?index ) .
} ORDER BY ?index
```

</td><td>

```
PREFIX co: <http://purl.org/co/List>
PREFIX : <http://example/>
SELECT ?item {
  :jane :likes [ co:item [
      co:itemContent ?item ; co:index ?index ]
  ] .
} ORDER BY ?index
```

</td></tr>
</table>

Other writings about RDF lists

- Updating RDF Lists with SPARQL, Any Seaborne
- RDF lists and SPARQL, Bob DuCharme