Versa

Mike Olson (Fourthought, Inc.), Uche Ogbuji (Fourthought, Inc.)
Revision (Initial release) [MO]

Versa is a specialized language for addressing and querying an RDF model. It allows traversal of arcs, processing of node contents, and general expression evaluation.

---

1. Introduction

Versa is a specialized language for addressing and querying nodes and arcs in a Resource Description Framework (RDF) model. It uses a simple and expressive syntax, designed to be incorporated into other expression systems, including XML, where, for instance, Versa can be used in extension functions or attributes of extension elements that provide RDF-related capabilities. Versa operates on the abstract graph model of RDF, and not any particular serialization.

Where used in this document, the keywords "SHOULD", "MUST", and "MUST NOT" are to be interpreted as described in RFC 2119 [RFC2119]. However, for readability, these words do not appear in all uppercase letters in this specification.

Versa uses constructs from XML namespaces for convenient abbreviation of URIs. Within this document, in examples and other discussion, some prefixes are commonly used without being defined each time. In this document consider these prefixes to be bound to the following namespaces:

- **versa**: http://rdfinference.org/versa/0/2/
- **vsort**: http://rdfinference.org/versa/0/2/sort/
- **vtrav**: http://rdfinference.org/versa/0/2/traverse/
- **h**: http://rdfinference.org/eg/versa/humanitas
- **dc**: http://purl.org/dc/elements/1.1
- **daml**: http://www.daml.org/2001/03/daml+oil#

This is just a convenience for this document. There is no normative binding of these prefixes.

2. The domain of operation

Versa queries operate on an RDF model. Any RDF model that follows the RDF Model defined in RDF 1.0 [RDFMS], or the most recent RDF Model Theory specification [RDFMT] is a valid processing space for Versa. There are some cases where the precise behavior of Versa is dependent on implementation details of the RDF model.

[DAML+OIL], while mostly based on RDF 1.0, does include some slight modifications and enhancements to RDF semantics. None of these changes appear to affect the abstract model, so Versa can also process DAML+OIL systems.

3. Data model

Versa operates on the abstract graph model of RDF. As such it operates on labeled nodes and arcs. In support of this processing, Versa defines a small set of standard data types.

3.1. Data types

3.1.1. Resource

A resource is a special string-like object that represents the URI of a resource in the model. Its literal expression can be in one of two forms. The first is as a simple QName (as defined in [XMLNS]), in which case the URI mapped to the prefix used in the QName is expanded to a string and concatenated to the local portion of the QName to derive the resource object's URI. Versa does not define a mechanism for mapping prefixes to URIs. Such a facility must be provided by a Versa implementation. For instance, when Versa is expresed within an XML document, prefixes might be mapped according to the namespace declarations in scope of the relevant element. A resource can also be expressed using the full URI in string form preceded by an "@" sign. This form is not a true literal: the @ is techically a conversion operator that take a string and return a resource object (See the section on conversions below.)

The following are examples of literal resources:

spam:eggs	if the prefix *spam* is mapped to the URI *http://python.org/*, the resulting resource has the URI *http://python.org/eggs*.
myobj:oute66	if the prefix *myobj* is mapped to the URI *urn:oid:this.is.not.really.a.valid.oid.r*, the resulting resource has the URI *urn:oid:this.is.not.really.a.valid.oid.route66*.
@"http://rdfinference.org"	A resource with URI *http://rdfinference.org*

Note that the lexical rules of XML QNames may limit the situations in they may be used in Versa. For instance, if the URIs are UUIDs in URN form. The full form can always be used to express any valid URI.

3.1.2. String

A sequence of zero or more characters, as defined in the XML 1.0 recommendation. Versa strings are similar to XPath strings. The main difference is in quote escaping.

Literal strings again are are expressed as in XPath: using either single or double quotes. The following are examples of literal strings:

"What thou lovest well remains, the rest is dross"	
'What thou lovest well remains, the rest is dross'	Equivalent to the above
"Embedded'Apostrophe"	If the string contains an apostrophe or quotaion mark, you would usually use the other to delimit the string.
"Embedded\"Apostrophe"	You can also use "\" to escape quotes
"Doubly'Embedded\"Quote'and\'Apostrophe"	Escaping allows you to express all variety of quote combinations within a string.

3.1.3. Number

Versa numbers are the same as XPath numbers: positive or negative floating-point numbers, based on the rules and semantics for double precision, 64-bit numbers in IEEE 754.

The following are examples of literal numbers:

2	Note that this is neither stored nor processed as an integer, since it is actually a floating point number expressed in abbreviated form.
3.14	pi to three decimal places
6.022e23	Avogadro's number: an example of using scientific notation

3.1.4. Boolean

Boolean types represent logical truth or falsehood. As such there are two boolean literals: `true` and `false`. `*` is provided as a synonym for `true`: a more readable form in such cases as traversal and filter expressions.

3.1.5. List

An heterogeneous ordered collection of any data type (including other lists or sets). Duplicates are allowed.

There is strictly no list literal. Lists can be expressed using the list() conversion function (see below) with simple values as the arguments. The following are examples of lists:

list("W. B. Yeats", "T. S. Eliot")	
list(2000, 2001, 2000)	
list(x:epound, x:tseliot, @"http://rdfinference.org/eg/versa/wyeats")	
list()	An empty list

3.1.6. Set

An heterogeneous unordered collection of any data type (including other sets or lists), with no duplicate values.

There is strictly no set literal. Sets can be expressed using the set() conversion function (see below) with simple values as the arguments. The following are examples of sets:

set("J. Alfred Prufrock", "Hugh Selwyn Mauberley")	
set(4.7, "four point seven")	
set(4.7)	Just one item

3.2. Conversions

When implicit conversions between data types are needed, the following matrix defines the operations applied.

From/To	Resource	String	Number	Boolean	List	Set
Resource	Identity	A resource with a URI as given by the string, using escaping as required [once a standard for RDF data literals emerges, using such would be the preferred approach]	If the resource can be interpreted as a representation of a number by the implementation, this is used, otherwise NaN.*	If the resource can be interpreted as a representation of a boolean by the implementation, this is used, otherwise false [Editor's note: this is a dangerous conversion, as the "exception" value lies within the normal boolean value space].*	List of length one with the resource in it.	Set of size one with the resource in it.
String	A resource with a URI as given by the string, using escaping as required	Identity	The number that is represented by the string, or NaN	false if the string is empty, otherwise true	List of length one with the string in it.	Set of size one with the string in it.
Number	A conversion of the number to a literal in URI form, as determined by the implementation*	String representation of the number	Identity	false if the number is positive or negative 0, otherwise true	List of length one with the number in it.	Set of size one with the number in it.
Boolean	A conversion of the boolean to a literal in URI form, as determined by the implementation*	"true" or "false"	0 if false, or 1 if true	Identity	List of length one with the boolean in it.	Set of size one with the boolean in it.
List	The result of conversion to resource of the first item in the	The result of conversion to string of the first item in the list, or "" if the list	The result of conversion to number of the first item in the	false if the list is empty, otherwise true	Identity	A set with the same entries as the list, except that if there are duplicate values, any equivalent

	if the list is empty	is empty	list, or 0 if the list is empty			are omitted (e.g. set(list(1,2,1)) = set(1,2))
Set	The result of conversion to list and then conversion to resource	The result of conversion to list and then conversion to string	The result of conversion to list and then conversion to number	The result of conversion to list and then conversion to boolean	A list with the same entries as the set, in arbitrary order	Identity

* Note that conversions from literals to URIs are especially subject to change as the RDFCore working group works on URI-based datatype literal representations

3.3. Conversion functions

Conversion functions are special functions that convert their arguments to a particular data type, using the conversion rules described above.

3.3.1. list

`list(`*`expression`*`[, `*`expression`*`, [...]])`

Create a list comprising each of the arguments in order.

3.3.2. set

`set(`*`expression`*`[, `*`expression`*`, [...]])`

Create a set comprising each of the arguments with duplicate values removed.

3.3.3. boolean

`boolean(`*`expression`*`)`

Return the boolean value of the argument

3.3.4. string

`string(`*`expression`*`)`

Return the string value of the argument

3.3.5. number

`number(`*`expression`*`)`

Return the number value of the argument

3.4. Comparisons

Comparisons between values follow specific rules for each data type. In general if two values are being compared, A and B, B is first converted to the same type as A before the comparison is made.

Comparisons can be explicitly applied in various operations, or can be explicitly made by invoking the relational functions (see below).

The following are the comparison rules for the various types.

Resources	Resources are converted to strings for comparisons.
Strings	The strings are compared as are XPath strings.
Numbers	The numbers are compared as are XPath numbers
Booleans	true is evaluated as greater than false.
Lists	Unspecified [Editor's note: suggestions are requested. On epossibility is the XPath approach of testing whether there is any intersection between the two lists. Or one could compare each list item in turn. Or one could compare the first items of each]

Sets Unspecified [Editor's note: suggestions are requested.]

3.4.1. Comparison or relational Functions

3.4.1.1. lt

`lt(`*`expression`* `[ , ` *`expression`* `])`

If there is a single argument, return true if the context is less than the argument (after the argument has been converted to the same type as the context). If there are two arguments, return true if the first is less than the second (after the second argument has been converted to the same type as the first). Otherwise return false.

3.4.1.2. gt

`gt(`*`expression`* `[ , ` *`expression`* `])`

If there is a single argument, return true if the context is greater than the argument (after the argument has been converted to the same type as the context). If there are two arguments, return true if the first is greater than the second (after the second argument has been converted to the same type as the first). Otherwise return false.

3.4.1.3. lte

`lte(`*`expression`* `[ , ` *`expression`* `])`

If there is a single argument, return true if the context is less than or equal to the argument (after the argument has been converted to the same type as the context). If there are two arguments, return true if the first is less than or equal to the second (after the second argument has been converted to the same type as the first). Otherwise return false.

3.4.1.4. gte

`gte(`*`expression`* `[ , ` *`expression`* `])`

If there is a single argument, return true if the context is greater than or equal to the argument (after the argument has been converted to the same type as the context). If there are two arguments, return true if the first is greater than or equal to the second (after the second argument has been converted to the same type as the first). Otherwise return false.

3.4.1.5. eq

`eq(`*`expression`* `[ , ` *`expression`* `])`

If there is a single argument, return true if the context is equal to the argument (after the argument has been converted to the same type as the context). If there are two arguments, return true if the first is equal to the second (after the second argument has been converted to the same type as the first). Otherwise return false.

3.4.1.6. neq

`neq(`*`expression`* `[ , ` *`expression`* `])`

If there is a single argument, return true if the context is unequal to the argument (after the argument has been converted to the same type as the context). If there are two arguments, return true if the first is unequal to the second (after the second argument has been converted to the same type as the first). Otherwise return false.

4. Versa Query Structure

4.1. Query

Versa defines expressions. An expression is a combination of literals, traversals and filters, variable references and function calls. Traversala and filters are expressions that matche patterns in the RDF model by selecting sets of starting resources and arc resources, and conditions for selecting end-points from the RDF model.

4.2. Context

Many Versa constructs are evaluated with regard to a context. The context is a value of any data type, and it can always be referred to explicitloy

in an expression using the token "."

4.3. Traversal and filter expressions

Traversal and filter expressions are the core of Versa. They provide a system for matching patterns in an RDF model by specifying desired nodes and arcs in the graph representing the model. The traversal and filter operators are the bases of the respective expression, and result in a list.

4.3.1. Forward traversal and filter expressions

The forward traversal operator matches patterns based on given sets of subjects and predicates. It returns a list of resulting objects. It takes the following form:

```
list - list -> boolean
```

The forward filter operator matches patterns based on given sets of subjects and predicates. In contrast to the forward traversal operator, it returns a list of the subjects that result from the patterns rather than the objects. It takes the following form:

```
list |- list -> boolean
```

In both cases the first *list* is a evaluated to obtain list of resources which are the subjects of statements. Each of these resources is set as the context for evaluating the second *list*, which is treated as a list of predicate resources. All statements in the model with these subjects and objects are marked as candidate statements. The object of each of the candidate statements is evaluated as the context of the *boolean*.

In the case of forward traversal, if the result, after conversion to boolean type, of evaluating the *boolean* is true, the object is added to the list of results. In the case of forward filtering, if the result, after conversion to boolean type, of evaluating the *boolean* is true, the subject of the corresponding statement is added to the list of results.

Unless an ordering aggregate function (see below) is used, the order of the resulting elements in the list is undefined in Versa, and is determined by the underlying model.

[*Editor's note:* When data types are formally incorporated into the Versa model, the treatment of object set expressions will certainly change]

4.3.2. Backward traversal expression

The backward traversal operator is similar to the forward traversal operator, but it is used to match patterns using the inverses of predicates. A backward traversal expression takes the following form:

```
list <- list - boolean
```

The first set is a set of resources which are the objects of statements, the predicates of which are given by the resources in the second set expression. This results in a list of matching statements, and the subject of each statement is evaluated as the context of the boolean expression. If the result, after conversion to boolean type, is true, the subject is added to the list of results. Conversions are automatically applied, as with forward traversal expressions.

Editor's note: There is no backward filter expression yet, but this post points to a possible use case.

4.3.3. Aggregate functions

Versa provides a set of special functions which are designed to be used within traversal operations to transform partial results within the context of the traversal.

4.3.4. sortq

```
sortq(set, expression [, vsort:number | vsort:string [, vsort:ascending | vsort:descending ] ])
```

Editor's note: Are there any others that cannot be expressed as operations on the result list? Grouping primitives, perhaps.

Editor's note: How to express multi-key sorting?

5. Variables

A variable reference evaluates to the value to which the variable name is bound in the set of variable bindings in the context. It is an error if the variable name is not bound to any value in the set of variable bindings in context.

6. Formal Versa grammar

```
[1] query ::= expression
[2] expression ::= traversal filter
                 | '(' expression ')'
                 | function-call
                 | literal
                 | particle
                 | variable-reference
[3] traversal ::= forward-traversal | backward-traversal
[3] filter ::= forward-filter
[4] forward-traversal ::= set-expression "-" set-expression "->" boolean-expression
[4] forward-filter ::= set-expression "|-" set-expression "->" boolean-expression
[5] backward-traversal ::= expression "<-" expression "-" expression
[6] function-call ::= identifier '(' ( expression ( ',' expression ) * ) ? ')'
[7] literal ::= resource-literal
              | string-literal
              | number-literal
              | boolean-literal
[8] resource-literal ::= '@' string-literal
[8] string-literal ::=
[9] number-literal ::=

[11] boolean-literal ::= 'true' | 'false'
[12] variable-reference ::= '$' identifier
[15] particle ::= '.' | '*'
```

7. Functions

Versa defines a core function library. Extension functions can be defined using a similar mechanism to that provided by XPath.

7.1. Resource functions

Resource functions operate on or return resources.

7.1.1. all

```
all([string, [string, [...]]])
```

Without any arguments, all returns a list of all resources in the model. If there are arguments, they are treated as query objects, evaluated and the results converted to boolean such that , it is a short cut for "

```
all(qo1, qo2, ..., qoN)
```

Is equivalent to

```
filter(all(), qo1, qo2, ..., qoN)
```

7.1.2. type

```
type(resource)
```

Returns a list of all resources of a specified type, as defined by RDFS and optionally DAML schema specifications. This function is essentially a short cut for:

```
all() - rdf:type -> *
```

7.1.3. traverse

```
traverse(list, list [ , vtrav:forward | vtrav:inverse [ , vtrav:transitive ] ])
```

This is a function similar to the traversal operator. The first and second arguments are converted to sets of resources. A list is generated with the results of matching all statements in the model with the subject given by the first argument and the predicate given by the second. The return value is a set of subjects or objects of the matching statements according to whether forward or inverse traversal has been selected. The special resource vtrav:any acts as a wild-card and causes any value to be matched.

The optional third argument indicates the direction in which predicates are to be traversed. It must match one of the following resources:

- *vtrav:forward*: return the objects of matching statements (the default)
- *vtrav:inverse*: return the subjects of matching statements

The optional fourth argument, which must match vtrav:transitive, traverses predicates transitively, considering all nodes along the chain. If omitted, no transitivity is applied except any applied by the underlying model (e.g. if rdf:type statements are traversed, or statements marked as daml:transitiveProperty).

7.1.4. properties

```
properties(list [ , vtrav:inverse ])
```

The first argument is converted to a set of resources (the source set), and a set of resources is returned. If the optional second argument is present, the resources are all properties expressed on each resource in the source set. Otherwise the returned properties are all those which have one of the source resources as a value.

7.2. Set and list functions

Set and list functions operate on or return sets and lists.

7.2.1. member

```
member(list [ , value ])
```

Return true if the list value of the first argument has an entry equal to the second. If only one argument is given, the context is used as the first.

7.2.2. distribute

```
distribute(list,
           string,
           [string, [...]])
```

distribute converts the first argument into a list. The second and subsequent arguments (the query arguments) are strings that are interpreted as Versa queries. It uses each item in the list as the context for evaluating each of the query arguments. The result is a list of lists; each entry in the outer list is a list containing the results from evaluating each of the query arguments in order using the Nth list item as context.

For example, the query:

```
distribute(list(@"http://4suite.org", @"http://rdfinference.org"), '.', 'string-length()', 'substring-after(., ":")')
```

returns

```
[[@"http://4suite.org", 17, "4suite.org"], [@"http://rdfinference.org", 23, "rdfinference.org"]]
```

The outer list is of length two because there are two items in the first argument. Each inner list has three items because there are so many query arguments.

7.2.3. map

```
map(string, list, [list, [...]])
```

map converts the first argument to string and dynamically evaluates this as a Versa query with one or more lists as the context. These lists are constructed as follows: The first item from each of the list expressions in the second and subsequent arguments are gathered into a list, as long as at least one list from the list expression arguments has an item. Then the second item if taken from each list expression, if at least one of them has two or more items, and so on, with as many iterations as the longest list from the list expression arguments. If the lists from the list expression arguments are of differing lengths, then all lists that are shorter than the longest are padded with nil resources (daml:nil).

The result is a list of values, as long as the longest item in the list from the list expression arguments.

As an example, the query:

```
map("concat()", ["A", "B", "C"], ["1", "2", "3"])
```

Will return a list of length 3:

```
["A1", "B2", "C3"]
```

And the query:

```
map(".-h:formatted-name->*", h:principia-h:author->*)
```

Returns the formatted name of the author of the book identified as "h:principia", and thus in our sample model would return

```
["Isaac Newton"]
```

This is equivalent to the chained traversal expression

```
h:principia - h:author -> * - h:formatted-name -> *
```

7.2.4. filter

```
filter(list, string, [string, [...]])
```

filter converts the first argument into a list (the source list). The second and following arguments are strings that are dynamically evaluated as Versa queries. The context for these evaluations are the items from the source list taken in turn. If all these evaluations return true, then the resource is added to the result list.

7.2.5. sort

```
sort(list[, conversion-indicator[, direction-indicator[, string ]]])
```

The argument is converted to a list or a set. The result is the list obtained by sorting according to the given criteria. The second parameter indicates the conversion that should be applied to each item before sorting, and the style of the resulting sort. It must be a resource with one of the following URIs:

- *vsort:string*: convert to string and sort according to unicode sorting conventions, as described in the XSLT specification's section on xsl:sort
- *vsort:number*: convert to number and sort according to the magnitude of the number

The default is vsort:string. The third parameter indicates the direction of sorting. It is converted to resource and must have one of the following URIs:

- *vsort:ascending*: sort in ascending order
- *vsort:descending*: sort in descending order

The default is http://rdfinference.org/versa/sort/ascending.

7.2.6. max

```
max(list[, conversion-indicator[, string ]])
```

The argument is converted to a list or a set (the source list). The result is the maximum value in the source list according to the given criteria. The second parameter indicates the conversion that should be applied to each item before sorting, and the style of the resulting sort. It must be a resource with one of the following URIs:

- *http://rdfinference.org/versa/sort/string*: convert to string and sort according to unicode sorting conventions [provide reference]
- *http://rdfinference.org/versa/sort/number*: convert to number and sort according to the magnitude of the number

The default is http://rdfinference.org/versa/sort/string.

max($a, $b, $c) is equivalent to head(sort($a, $b, v:descending, $c))

7.2.7. min

```
min(list[, conversion-indicator[, string ]])
```

The argument is converted to a list or a set (the source list). The result is the minimum value in the source list according to the given criteria. The second parameter indicates the conversion that should be applied to each item before sorting, and the style of the resulting sort. It must be a resource with one of the following URIs:

- *http://rdfinference.org/versa/sort/string*: convert to string and sort according to unicode sorting conventions [provide reference]
- *http://rdfinference.org/versa/sort/number*: convert to number and sort according to the magnitude of the number

The default is http://rdfinference.org/versa/sort/string.

min($a, $b, $c) is equivalent to head(sort($a, $b, v:ascending, $c))

7.2.8. union

`union(`*`set`*`, `*`set`*`)`

Both arguments are converted to sets, and the result is a set consisting of all items that are in either argument set.

7.2.9. intersection

`intersection(`*`set`*`, `*`set`*`)`

Both arguments are converted to sets, and the result is a set consisting of all items that are in both argument sets.

7.2.10. difference

`difference(`*`set`*`, `*`set`*`)`

Both arguments are converted to sets, and the result is a set consisting of all items that are in neither argument set.

7.2.11. join

`join(`*`list`*`[, `*`list`*`, [...]])`

Each argument is converted to a list, and the result is a list which consists of the concatenation of all the argument lists in order.

7.2.12. head

`head(`*`list`*`, [`*`number`*`])`

Converts the first argument to a list L, and the second to a number N. Returns a list consisting of the first N items in L. N defaults to 1. If N is negative, or exceeds the length of the list, the entire list is the result.

7.2.13. rest

`rest(`*`list`*`, [`*`number`*`])`

Converts the first argument to a list L, and the second to a number N. Returns a list consisting of all items in L after position N. N defaults to 1. If N is negative, or exceeds the length of the list, an empty list is the result. The following expression returns the same list as L, regardless of the value of N:

```
concat(first(L, N), rest(L, N))
```

7.2.14. tail

`tail(`*`list`*`, [`*`number`*`])`

Converts the first argument to a list L, and the second to a number N. Returns a list consisting of the last N items in L. N defaults to 1. If N is negative, or exceeds the length of the list, an empty list is the result.

7.2.15. length

`length(`*`list`*`)`

Converts the argument to a list and returns the number of items in the list.

7.2.16. slice

`slice(`*`list`*`, `*`number`*` [, `*`number`*`])`

The first expression is evaluated and converted to a list (the source list). The second argument is evaluated and converted to a number which is

the starting index. If the third argument is present, it is evaluated and converted to a number which is the ending index. If not specified, the ending index is the length of the list. Return a new list comprising the elements in the source list from the starting index to the ending index, in order.

7.3. Number Functions

Number Functions are functions that work with numbers. All return number types

Editor's note: the Versa number function library is identical to that of XPath. The details will be adde din the next draft.

7.4. String Functions

String functions work with strings.

7.4.1. concat

```
concat(string[, string, [...]])
```

Each argument is converted to a string, and the result is a string which consists of the concatenation of all the arguments in order.

7.4.2. starts-with

```
starts-with(string [ , string ])
```

Return true if the string value of the first argument starts with the value of the second. If only one argument is given, the context is used as the first argument.

7.4.3. contains

```
contains(string [ , string ] [ versa:ignore-case ])
```

Return true if the string value of the second argument is a substring value of the first. If only one argument is given, the context is used as the first.

7.4.4. substring-before

```
substring-before(string [ , string ])
```

Convert all arguments to strings. Return the substring of the first argument that precedes the first occurrence of the second argument, or the empty string if the first argument does not contain the second. If only one argument is given, the context is used as the first argument.

7.4.5. substring-after

```
substring-after(string [ , string ])
```

Convert all arguments to strings. Return the substring of the first argument that succeeds the first occurrence of the second argument, or the empty string if the first argument does not contain the second. If only one argument is given, the context is used as the first argument.

7.4.6. substring

```
substring(string-or-number, number [, number])
```

The first argument is the source string and the second is the starting index. If there is a third argument, it is the ending index, otherwise the ending index is the length of the string. The substring is returned comprising the characters in the source string from the starting index to the ending index. If the ending index exceeds the length of the string, the length of the string is substituted for its value.

7.4.7. string-length

```
string-length([string])
```

If there is no argument, the context is converted to a string and used as the argument. The number of characters in the string is returned.

7.4.8. find-regex

```
find-regex(string [ , string ] [ versa:ignore-case ])
```

If there are three arguments, then after conversions, the first is the outer string, the second the inner string and the last one must be the special flag resource versa:ignore-case. If there are two arguments and the second can be converted to the special flag resource versa:ignore-case, then the first one is the inner string and the context is the outer string. If there are two arguments and the second cannot be converted to the special flag resource versa:ignore-case, then the first one is the outer string and the second is the outer string. If there is only one argument, then it is the inner string and the context is the outer string.

The inner string is interpreted as a POSIX Simple Regular Expression [PSRE] and The return value is the first index at which it is matched in the outer string. If the versa:ignore-case flag is given, then the regular expression is matched without regard to the case of alphabetic characters. If there is no regular expression match at all, the return value is -1. [*Editor's note:* is this a suitable flag for "not found"? NaN is another option, though perhaps a clumsy one.]

7.5. Boolean Functions

Boolean Functions are functions that work with booleans. All return number types of 0 or 1

7.5.1. and

```
and(boolean[, boolean, [...]])
```

Return true if the boolean values of all the arguments are true.

7.5.2. or

```
or(boolean[, boolean, [...]])
```

Return true if the boolean value of one or more arguments are true

7.5.3. not

```
not(boolean)
```

If the boolean value of the argument is true, return false, otherwise return true.

7.5.4. isResource

```
isResource([value])
```

Return true if the argument is a resource, or can be converted to a resource according to the determination of the underlying RDF model. If no argument is given, the context is used as the argument.

7.5.5. isLiteral

```
isLiteral([value])
```

Return true if the argument is not a resource, and can not be converted to a resource according to the determination of the underlying RDF model. If no argument is given, the context is used as the argument.

8. References and resources

[PSRE]: The Open Group UNIX Specification on Regular Expressions

[RFC2119]: RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels

[RDFMS]: Resource Description Framework (RDF) Model and Syntax Specification

[RDFMT]: RDF Model Theory

[RDFS]: Resource Description Framework (RDF) Schema Specification 1.0

[]:

9. Appendix A (non-normative): Use cases

[*Editor's note:* Yes, this section is in woeful need of work.]

In order to guide the development of Versa some use cases for RDF query have been developed. This section presents these use cases, as well as how they can be addressed using the current specification of Versa

9.1. Get all resources of type "h:Person"

Often one wants to simply check a model for all resources with a given RDF type

9.1.1.

Versa provides the `type` function to deal with this common case conveniently:

```
type(h:Person)
```

returns

```
set(h:epound, h:teliot, h:wyeats)
```

Or the using traversal expressions (and giving the same result):

```
all() - rdf:type -> h:Person
```

9.2. Get all people named "Ezra Pound"

9.2.1.

```
type(h:Person) - h:formattedName -> eq("Ezra Pound")
```

which results in

```
set(h:epound)
```

There are alternative ways to express this. For instance, using abbreviated forward traversal and filters:

```
filter(h:formattedName(type(h:Person)), eq("Ezra Pound"))
```

9.3. The name and age of the oldest author

9.3.1.

```
distribute(max(all() - h:author -> *, v:number, h:age()), 'h:age()', 'h:formattedName()')
```

9.4. The name of the second oldest ancestor of Joe

A DAML-aware Versa implementation will allow easy querying of explicitly transitive properties, but often one needs to interpret properties tarnsitively without help from the schema.

9.4.1.