ajp — A JSONPATH processor for RFC9535 implemented in Invisible XML, XSLT and XPath

Alan Painter `<contact@xmljacquard.org>`

7 November 2025

Revision History

Revision 1.1	27 November 2025
minor corrections	
Revision 1.0	7 November 2025
initial	

Abstract

Herein is described an implementation of RFC9535, also known as JSONPATH, using Invisible XML, XSLT and XPath with the intention of providing both a reasonable, conformant RFC9535 implementation and to showcase the expressiveness of XSLT/XPath combined with Invisible XML.

Table of Contents

1. Proposing an XSLT/XPath/Invisible XML implementation of RFC9535

In February 2024, with the advent of RFC9535 [RFC9535], the proposed standard for the JSONPATH query language, the Internet Engineering Task Force (IETF) has standardized the syntax and semantics of a language which, since its original proposal in 2007 [JPTHORIG], has been widely adopted with at least 50 different implementations [BURGCMP]. The divergence of the results of the different implementations is one of the motivations for standardizing with RFC9535.

There are a number of different objectives for proposing an implementation of RFC9535 in XSLT/XPath, including:

- Produce an implementation with strict and demonstrable conformance to RFC9535.
- Introduce an additional example of using Invisible XML, in this case for processing a grammar that was originally provided as an ABNF grammar.
- Showcase some of the advantages of XSLT [XSLT30]/XPath [XPATH31] in implementing RFC9535.
- Provide an implementation of RFC9535 that can be re-used from the different host languages that support Invisible XML, XSLT and XPath.
- Gain some additional insight into the RFC9535 specification through implementation.

2. ajp — A JSONPATH Processor

The implementation source code for *ajp* — A JSONPATH Processor — can be found at: https://github.com/xmljacquard/ajp

This document describes the release version **0.0.3** of *ajp*.

3. Forewarning on an abuse of XPath notation

For facility of presentation, especially in terms of width of lines of code, many XPath functions that are implemented in the repo via XSLT (i.e. using `xsl:function` elements) are presented here in a *pseudocode* that itself is not valid XPath. For instance, the following XPath function implemented in XSLT:

```
<xsl:function name="ajp:applySegments" as="map(xs:string, item()?)*"             >
    <xsl:param name="root"             as="item()?"                             />
    <xsl:param name="segments"         as="map(xs:string, array(function(*))+ )*" />

    <xsl:sequence select="let $startNodelist  := map { '$' : $root },
                              $returnNodelist := ajp:applySegments($startNodelist, $segments,
                          return ajp:convertNulls($returnNodelist)" />
</xsl:function>
```

will be presented as

```
function ajp:applySegments($root     as item()?,
                           $segments as map(xs:string, array(function(*))+ )*
                          )          as map(xs:string, item()?)*
{
    let $startNodelist  := map { '$' : $root },
        $returnNodelist := ajp:applySegments($startNodelist, $segments, $root)
    return ajp:convertNulls($returnNodelist)
}
```

The function name (i.e. `ajp:applySegments`) in the pseudocode is not valid XPath because XPath itself does not have a mechanism for creating named functions; nonetheless, the intended meaning should be clear enough and, as the code text is greatly shortened by this artifice, this paper will employ this abusive notation throughout.

4. RFC9535 basics

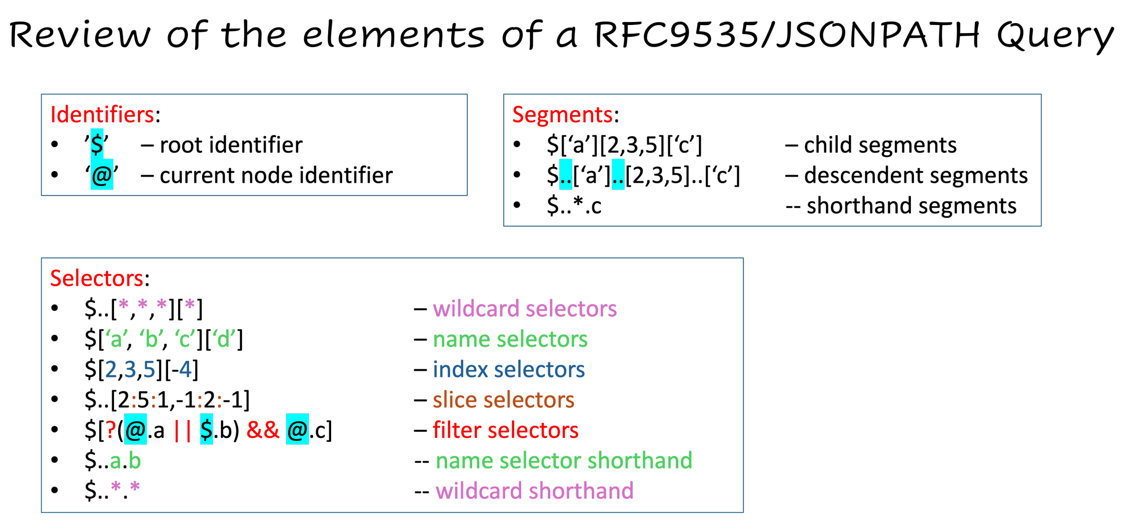
4.1. XMLPrague 2024 Presentation

For more detailed information concerning JSONPATH and its definitions, please refer to the XML Prague 2024 presentation *JSONPath: an IETF Proposed Standard, with comparisons to XPath* [PRAG2024]. This paper will make extensive use of JSONPATH vocabulary which is described in that paper.

4.2. Identifiers, Segments and Selectors

Section 1.4 of RFC9353 [RFC9535] gives an overview of the elements of a JSONPATH query, notably as Identifiers (2), Segments (2 types) and Selectors (5 different selectors). The Identifiers are only used within filter selectors, with the exception of the obligatory first character of any JSONPATH expression that must be the `root-identifier`, the character `'$'`. Following the `'$'` char is a possibly-empty sequence of *segment*s, with each *segment* being either a *child* or a *descendant* segment. Finally, each *segment* is composed of one or more *selector*s. This is the very simple structure of a JSONPATH query string.

Figure 1. Elements of a JSONPATH query



Review of the elements of a RFC9535/JSONPATH Query

Identifiers:

• '$'	– root identifier
• '@'	– current node identifier

Segments:

• $['a'][2,3,5]['c']	– child segments
• $..['a']..[2,3,5]..['c']	– descendent segments
• $..*.c	-- shorthand segments

Selectors:

• $..[*,*,*][*]	– wildcard selectors
• $['a', 'b', 'c']['d']	– name selectors
• $[2,3,5][-4]	– index selectors
• $..[2:5:1,-1:2:-1]	– slice selectors
• $[?(@.a || $.b) && @.c]	– filter selectors
• $..a.b	-- name selector shorthand
• $..*.*	-- wildcard shorthand

4.3. The SYNTAX of JSONPATH/RFC9535

The details of the JSONPATH query expression are largely described in the RFC9535 ABNF (Augmented Backus Naur Form) grammar [RFC5234] which can be used to "parse" the query expression. This grammar is provided in its entirety in Appendix A of RFC9535 [RFC9535].

4.4. The Semantics of JSONPATH/RFC9535

Whereas the ABNF grammar description can be considered to be the syntax of RFC9535, the semantics of the JSONPATH standard are described in other sections of RFC9535, specifically in the sections labelled "Semantics" (i.e. 2.2.2, 2.3.1.2, 2.3.2.2, 2.3.3.2, etc). The semantics are described at length in RFC9535. The experience of developing *ajp* has shown a few spots where the non-normative JSONPATH Compliance Test Suite [JPTHCTS] greatly clarifies the descriptions in RFC9535. A specific case is described below.

5. A description of *ajp*

5.1. An XSLT package

The current implementation of *ajp* (i.e. version **0.0.3**) is contained within an XSLT 3.0 package. The component that uses *ajp* must also be XSLT 3.0.

The implementation repo also includes a Java API for calling *ajp* and retrieving the results.

In general, an XSLT package is meant to be usable from any XSLT environment and host language. In the case of the version **0.0.3** of *ajp*, the XPath extension functions necessary for the Invisible XML (*ixml*) parsing are currently available only in Java.

5.2. XPath3.1 and JSON

Since version 3.1, the XPath Data Model (XDM, [XDM31]) contains types for arrays and for maps, where the JSON object type is the equivalent of an XPath map with a string key. The XPath 3.1 data model can represent JSON as well as the XML data types from previous versions of XDM.

5.3. The fundamental structures used in *ajp*

The XPath definitions for *segments* and *nodelists* are fundamental data structures for the implementation.

5.3.1. Nodelist

In section 1.1 Terminology, RFC9535 defines a *Nodelist* as an ordered list of *Nodes*. A *Node* itself is described as a pair of values: a String representing the location of the *Node* within the Query Argument and the JSON value at that location in the Query Argument.

RFC9535 does not define the concrete structure for the *Node* and the *Nodelist*, leaving that to the implementation. Within *ajp* and using XPath, a *Node* is defined as a singleton map (i.e. a map with a single entry) with a `xs:string` key and an `item()?` value. A *Nodelist* is an XPath sequence of Nodes.

An example of values in a *Nodelist* can be seen below. It is important to note that the *Nodelist* is used as an input as well as an output of segment processing. In addition, a *Nodelist* is used as an intermediate value for comparison and test expressions within a `filter-selector`.

Figure 2. The XPath Nodelist Structure

```
$nodelist as map(xs:string, item()?)*

map { “$[‘a’]”, 1 }                    map { “$”, ** }
map { “$[‘e’]”, [42, 23] }
map { “$[‘e’][0]”, 42 }
```

5.3.2. Segments

RFC9535 shows the general structure of a JSONPATH query to be the `root-identifier` (or `'$'`) followed by a sequence of segments, possibly an empty sequence, for which each segment is composed of one or more selectors. The *ajp* implementation describes this as a sequence of singleton maps, each map having either the key `'child'` or `'descendant'` to indicate the type of segment, and having a non-empty sequence of arrays of two functions, one array per selector, as the value. This is the output form for the segments as produced by *ajp*'s query compilation. The segments produced by an example JSONPATH query is given below:

Figure 3. The XPath Segments Structure

```
$segments as map( xs:string, array(function(*))+ )*

$..[‘a’, ’e’, 0][1]

map { “descendant”, [  nameSelectorKeys(‘a’), nameSelectorTest()  ]
                    [  nameSelectorKeys(‘e’), nameSelectorTest()  ]
                    [  indexSelectorKeys(0),  indexSelectorTest() ]  }
map { “child”,      [  indexSelectorKeys(1),  indexSelectorTest() ]  }
```

5.3.3. Arrays of Functions for the Selectors

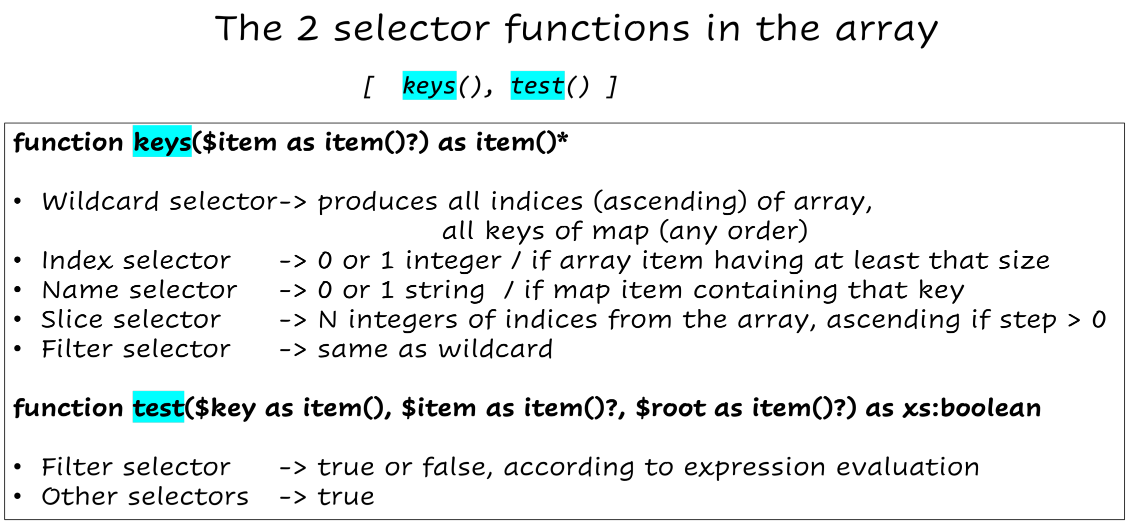
The values in the *segments* maps are ordered, non-empty sequences of arrays of functions, with one array for each selector in the segment. An array for a selector will have two functions, a *Keys()* function in the first position of the array and a *Test()* function in the second position of the array.

Since XPath 3.0, functions have been added to the XPath Data Model (XDM) data types, allowing functions to be arguments of functions, return values of functions and also data values. *ajp* is using an array to hold the two functions that are used to implement a given selector.

The *Keys()* and *Test()* selector functions have the below function signatures. The *Keys()* function takes, as its single argument `$item`, a JSON XDM value, and it returns the keys that correspond to that value for that selector. For instance, if the JSON value is an array of size 5 and the selector is the `index-selector` with index 4 (i.e. 0-based index in JSONPATH), then a call to the `Keys()` selector function on that array would return a single item, the value of the last element of the array. If the array had size 4, then the call to the selector function `Keys()` for the smaller array would return an empty sequence, because there is no element at 0-based index 4 in the array. If `item` is something other than an array, say a JSON object or a JSON primitive, then the index selector `Keys()` function would also return an empty sequence, since the item itself is not an array.

The *Test()* function takes three arguments, including a reference to the `$root` or query argument, and returns a boolean value to indicate if the node should be included in the output nodelist from that segment. Note that for all selectors other than the `filter-selector`, the *Test()* function returns a trivial `true()` value.

Figure 4. Signatures of the Selector Functions



The 2 selector functions in the array

[keys(), test()]

function keys($item as item()?) as item()*

```
• Wildcard selector-> produces all indices (ascending) of array,
                        all keys of map (any order)
• Index selector    -> 0 or 1 integer / if array item having at least that size
• Name selector     -> 0 or 1 string  / if map item containing that key
• Slice selector    -> N integers of indices from the array, ascending if step > 0
• Filter selector   -> same as wildcard
```

function test($key as item(), $item as item()?, $root as item()?) as xs:boolean

```
• Filter selector   -> true or false, according to expression evaluation
• Other selectors   -> true
```

5.4. Three phases of *ajp* evaluation

The *ajp* processor executes globally in three phases as described below. The first two phases correspond to the *compile time* phase and the third phase is the *run time* phase, which can be executed repetitively on different JSON values without requiring re-compilation for the same JSONPATH expression.

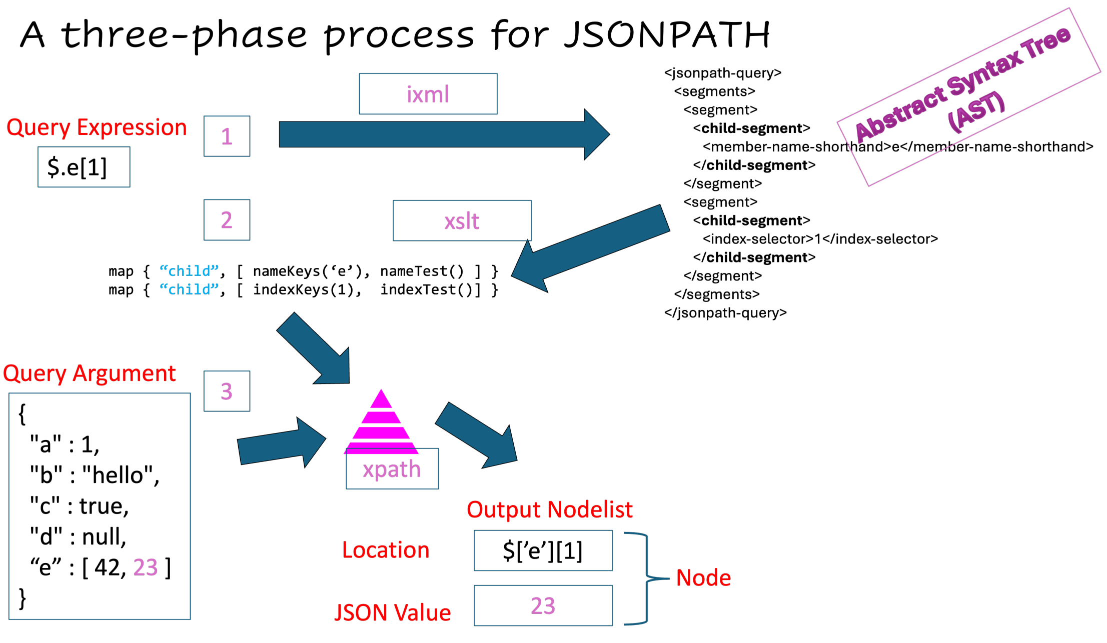
1. Phase 1 — Using Invisible XML (ixml), generate an XML document / Abstract Syntax Tree (AST) from the JSONPATH query expression.

2. Phase 2 — Using XSLT and the AST from Phase 1, generate the *segments* description of the query expression.

3. Phase 3 — Using the *segments* description from Phase 2 and given a JSON value, produce the resulting *nodelist* output.

The three phases are described in the following diagram.

Figure 5. Three phases of processing a JSONPATH expression on a JSON value


A three-phase process for JSONPATH
Query Expression
$.e[1]
1
ixml
<jsonpath-query>
<segments>
<segment>
<child-segment>
<member-name-shorthand>e</member-name-shorthand>
</child-segment>
</segment>
<segment>
<child-segment>
<index-selector>1</index-selector>
</child-segment>
</segment>
</segments>
</jsonpath-query>
Abstract Syntax Tree (AST)
2
xslt
map { "child", [nameKeys('e'), nameTest()] }
map { "child", [indexKeys(1), indexTest()] }
Query Argument
3
{ "a" : 1, "b" : "hello", "c" : true, "d" : null, "e" : [42, 23] }
xpath
Output Nodelist
Location
$['e'][1]
JSON Value
23
Node

5.5. Two XPath functions to encapsulate the three phases

The three phases described in the previous section are encapsulated within two XPath functions.

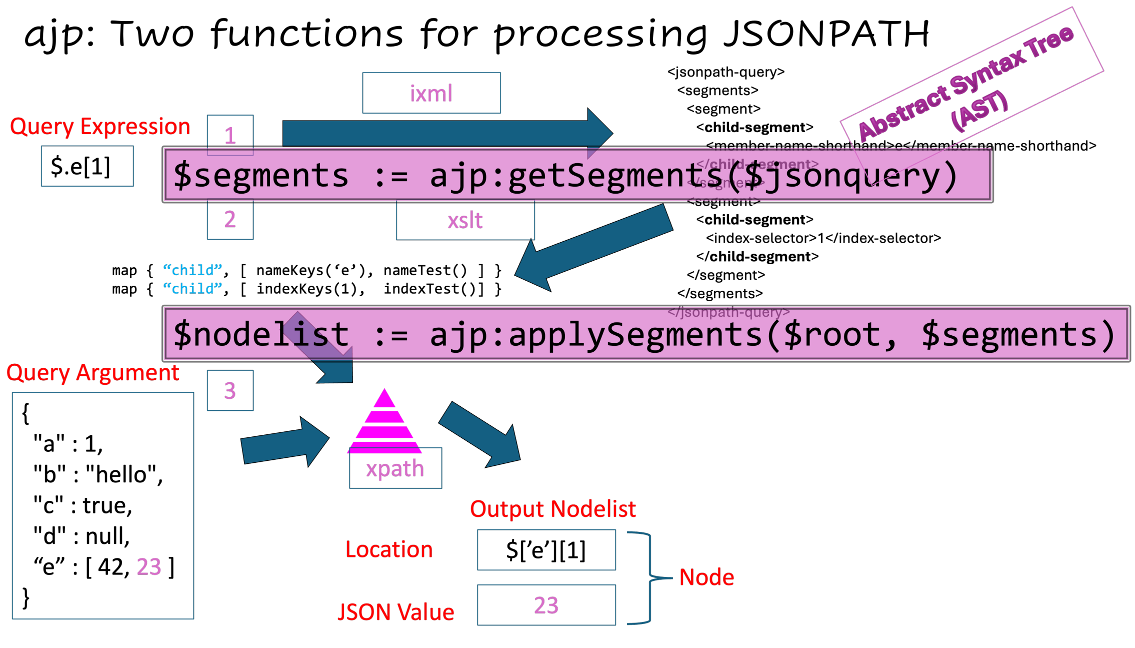
1. $segments := ajp:getSegments($jsonquery)

2. $nodelist := ajp:applySegments($root, $segments)

The function `ajp:getSegments()` accepts a JSONPATH string argument and returns a *segments* description. This function essentially is the *compilation* of the JSONPATH expression into a reusable form. In case of any error in the JSONPATH expression, an error is raised and the *segments* description is not created.

The function `ajp:applySegments()` takes two arguments: a JSON value (or Query Argument) and the *segments* description returned from previous function. `ajp:applySegments()` can be called multiple times in order to apply the same JSONPATH expression to different JSON values. This function is essentially the *runtime* phase of JSONPATH evaluation.

Figure 6. Two functions that encapsulate the three phases of processing a JSONPATH expression on a JSON value



```
$segments := ajp:getSegments($jsonquery)
```

```
<jsonpath-query>
  <segments>
    <segment>
      <child-segment>
        <member-name-shorthand>e</member-name-shorthand>
      </child-segment>
    </segment>
    <segment>
      <child-segment>
        <index-selector>1</index-selector>
      </child-segment>
    </segment>
  </segments>
</jsonpath-query>
```

```
map { "child", [ nameKeys('e'), nameTest() ] }
map { "child", [ indexKeys(1),  indexTest()] }
```

```
$nodelist := ajp:applySegments($root, $segments)
```

```
{
  "a" : 1,
  "b" : "hello",
  "c" : true,
  "d" : null,
  "e" : [ 42, 23 ]
}
```

5.6. Errors

RFC9535 Section 2.1 *Overview* indicates that any errors in the query expression must be discovered independently of any JSON value in the query argument. In the case of *ajp*, this means that the error must be discovered during the call to `ajp:getSegments()`. The implementation will raise an XPath error in this case. (See [XPATH31F] Section 3.1 *Raising errors*.)

RFC9535 also indicates that no errors can be raised other than errors in the validity of the query expression.

6. *ajp* Runtime

The evaluation phase of *ajp* is based upon the *segments* and *nodelist* structures described previously. In order to motivate the utility of the *segments* structure, the process of evaluating the resulting nodelist via the *segments* description will be described here first.

The evaluation phase is executed by calling `ajp:applySegments()` with the two arguments: the JSON query argument (or `$root`) and the *segments* description from `ajp:getSegment()`. The evaluation is a very simple implementation which requires only a handful of small XPath functions. These functions are detailed below.

6.1. Segment Processing

RFC9535 Section 2.1.2 "JSONPATH Syntax and Semantics -- Semantics" describes how segments are processed.

> "The query is a root identifier followed by a sequence of zero or more segments, each of which is applied to the result of the previous root identifier or segment and provides input to the next segment. These results and inputs take the form of nodelists."

> "The nodelist resulting from the root identifier contains a single node (the query argument). The nodelist resulting from the last segment is presented as the result of the query."

The Segment Processing is performed in *ajp* via a few short functions in XPath:

Figure 7. Two argument version of `ajp:applySegments()`

```
function ajp:applySegments($root     as item()?,
                           $segments as map(xs:string, array(function(*))+ )*
                          )          as map(xs:string, item()?)*
{
    let $startNodelist  := map { '$' : $root },
        $returnNodelist := ajp:applySegments($startNodelist, $segments, $root)
    return ajp:convertNulls($returnNodelist)
}

function ajp:convertNulls($nodelist as map(xs:string, item()?)*)
                                    as map(xs:string, item()?)*
{
    for $map in $nodelist
    return if ($map?* instance of node() and $map?* is $NULL)
           then map:entry(ajp:key($map), ())
           else $map
}
```

The two-argument version of `ajp:applySegments()` is called from the using application. Here, the initial `$nodelist` containing only the query argument (i.e. `$root`) is constructed and passed to the recursive, three-argument version of `ajp:applySegments()`, along with the *segments* description from `ajp:getSegments()`. As a third parameter, the `$root` is also passed so that it can be used within a `filter-selector` in case a *subquery* references the `root-identifier` or `'$'`. This use of `$root` will be addressed below.

Before returning the result `$nodelist` to the caller, any `$NULL` values are replaced with empty sequences by the function `ajp:convertNulls()`. The reason for the `$NULL` value is addressed below.

Figure 8. Three argument, recursive version of `ajp:applySegments()`

```
function ajp:applySegments($nodelist as map(xs:string, item()?)* ,
                           $segments as map(xs:string, array(function(*))+ )*,
                           $root     as item()?
                          )          as map(xs:string, item()?)*
{
    if (empty($segments))
    then $nodelist
    else let $resultNodelist :=
             if (ajp:key(head($segments)) eq 'child')
             then ajp:children   ($nodelist, head($segments)?*, $root)
             else ajp:descendants($nodelist, head($segments)?*, $root)
         return ajp:applySegments($resultNodelist, tail($segments), $root)
}
```

The three-argument version of `ajp:applySegments()` will apply the *selectors* of the head *segment* (i.e. **`head($segments)?*`**) to the incoming `$nodelist` by calling either the `ajp:children()` or the `ajp:descendants()` function to traverse the nodes in the `$nodelist`, apply this segment's *selectors* and then capture the head *segment*'s output as `$resultNodelist` in a `let` variable in order to then pass it to a recursive call to `ajp:applySegments()` in order to process the *selectors* of any remaining segments.

These functions implement RFC9535 Section 2.1.2.

6.2. Selector Processing

In order to implement the *selector* processing and traverse the nodes in the `$nodelist`, either the *children* or *descendants* according to the *segment* type, `ajp:applySegments()` calls either `ajp:children()` or `ajp:descendants()`. It is within this former function, `ajp:children()`, that all *selector* processing is performed.

RFC9535 Section 2.5 and 2.5.1.1 describe the output of *Nodes* in an output *Nodelist* as a function of the input *Nodes* in the input *Nodelist* as well as the *selectors* within a *segment*. In Section 2.5, describing in general both *child* and *descendant* segments:

> *For each node in an input nodelist, segments apply one or more selectors to the node and concatenate the results of each selector into per-input-node nodelists, which are then concatenated in the order of the input nodelist to form a single segment result nodelist.*

6.2.1. Child Segment Selector Processing

Section 2.5.1.2 Child Segment / Semantics goes on to describe the traversal of child nodes of the nodelist:

> *In summary, a child segment drills down one more level into the structure of the input value.*

The `ajp:children()` function performs this operation.

Figure 9. `ajp:children()`

```
 1 function ajp:children($nodelist  as map(xs:string, item()?)*,
                        $selectors as array(function(*))+,
                        $root      as item()?
                       )           as map(xs:string, item()?)*
 5 {
       for $node        in $nodelist [ ajp:isMapOrArray(.?*) ],
           $parentPath  in ajp:key($node),
           $parentValue in $node?*,
           $selector    in $selectors,
10         $key         in $selector(1)($parentValue),
           $path        in ajp:path($parentPath, $key)
       return map {
                    $path : ( $parentValue($key), $NULL )[1]
                } [ $selector(2)($key, $parentValue, $root) ]
15 }
```

The `ajp:children()` function is a single XPath `for` expression which iterates over all of the eligible *Nodes* in the input `$nodelist` (line 6) and then binds two variables (`$parenthPath`, `$parentValue`) to the location and value of the parent items for which we are selecting the children. Each `selector` in the segment will have its array of functions referenced such that `$keys` associated with each selector will be generated for that item by calling the *Keys()* function for that selector, which is the first function in the array, at position 1 (line 10). This key's value is then used to create the child's location by calling the function `ajp:path` (line 11). Finally, the `return` expression shows where the singleton map will be created with this child item's *location* and *JSON value* but only if the selector's *Test()* function, the function at position two, returns `true` (line 14).

One other point to mention is that the output *Node*'s value will not be the XPath empty sequence, but rather the XML element referenced by the variable `$NULL`. The reason is that this allows us to distinguish between a literal JSON `null` value and an XPath empty sequence when performing comparisons in the `filter-selector`.

This function, `ajp:children()`, is the only place where *ajp* will call the *selector* functions that make up the values of the *segments* structure.

6.2.2. Descendant Segment Selector Processing

RFC9535 Section 2.5.2.2 Descendant Segment / Semantics stipulates:

> *For each node in the input nodelist, a descendant selector visits the input node and each of its descendants such that:*
>
> - *nodes of any array are visited in array order, and*
> - *nodes are visited before their descendants.*

The function `ajp:descendants()` executes the *nodelist* traversal for the *descendant segment*.

Figure 10. `ajp:descendants()`

```
function ajp:descendants($nodelist  as map(xs:string, item()?)*,
                         $selectors as array(function(*))+,
                         $root      as item()?
                        )           as map(xs:string, item()?)*
{
    ajp:children($nodelist, $selectors, $root)
    ,
    for $node       in $nodelist [ ajp:isMapOrArray(.?*) ],
        $parentPath in ajp:key($node),
        $key        in ajp:simpleKeys($node?*),
        $childPath  in ajp:path($parentPath, $key),
        $childValue in $node?*($key)
    return ajp:descendants( map { $childPath : $childValue }, $selectors,
                                                              $root )
}

function ajp:simpleKeys($item as item()?) as item()*
{
    if ($item instance of map(*))
    then map:keys($item)
    else if ($item instance of array(*))
    then (1 to array:size($item))
    else ()
}
```

Here the function for processing selectors in child segments, `ajp:children()`, is first called on the entire *nodelist*, followed by a `for` expression that iterates over the nodes in the node list, generating the keys for any *map* or *array* children using `ajp:simplekeys()` and then effecting a recursive call to `ajp:descendants()` on each child of a *node* in the *nodelist*, passing the child *node*'s location and value as the sole node in the recursive *nodelist*. This will visit the parent nodes before the child nodes and hence implements Section 2.5.2.2.

6.2.3. `ajp:path()`

Both `ajp:children()` and `ajp:descendants()` use the function `ajp:path()` to create the location (i.e. the normalized JSONPATH expression) from the parent's location and the key that represents the child (integer or string).

Figure 11. `ajp:path()`

```
function ajp:path($path as xs:string, $key as item()) as xs:string
{
    if ($key instance of xs:string)
    then concat($path, '[', '''', ajp:escape($key), '''' ']')
    else if ($key instance of xs:integer)
    then concat($path, '[',       $key - 1,              ']')
    else ()
}
```

6.2.4. `ajp:escape()`

RFC9535 Section 2.7 describes how a location (i.e. a Normalized Path) must be encoded in order to be unique. Certain unicode characters (e.g. carriage return and newline) have escape sequences (e.g. `'\r'` and `'\n'` respectively). Other characters do not have common associated glyphs or escape sequences and are hence presented with their hex values (e.g. `'\u000b'` for LT or Line Tabulation. Because `ajp:path()` is used to create the location from the parent's location and the key that represents the child (integer or string), the key value is "normalized" via the `ajp:escape()`. For brevity, this function will not be presented here.

6.3. Conclusion: the *ajp* runtime is very simple

This section has presented the handful of small XPath functions that perform the "runtime" evaluation of a JSONPATH expression that has been "compiled" into a *segments* description. The brevity and simplicity of these functions give witness to the overall simplicity of the *runtime* evaluation with this *segments* description. This evaluation mechanism motivates the creation of the *segments* description.

7. *ajp* Compile time

7.1. Processing the JSONPATH query with Invisible XML (*ixml*)

Invisible XML (*ixml*) is a recent initiative that provides tools for describing structured data as a grammar and then serializing the description of the data into XML. [IXML10] *ajp* uses *ixml* for parsing the JSONPATH query as it is defined by the ABNF (Augmented Backus Naur Form) grammar [RFC5234] within RFC9535 (Appendix A). *ajp* uses in its first version the *ixml* processor *nineml* [NINEML] but with the expectation that other *ixml* processors will also be available in the future, as *nineml* is currently limited to Java as a host language.

7.1.1. Adapting ABNF to *ixml*

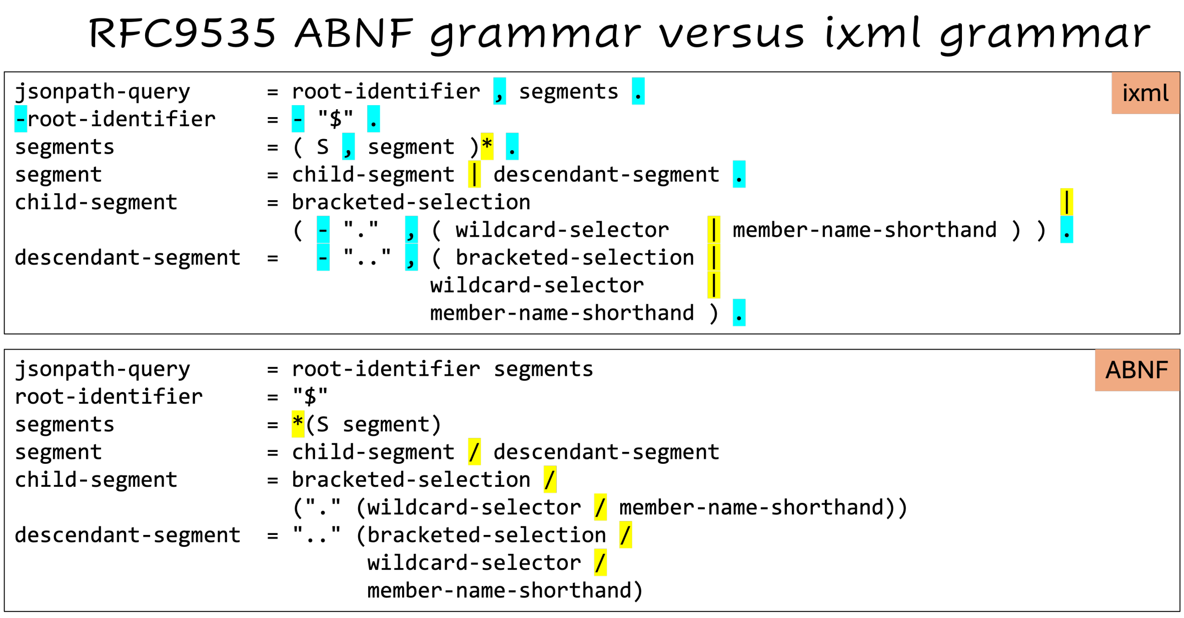
The RFC9535 grammar is around 150 lines of ABNF which was converted to around 200 lines of *ixml* for *ajp*. The increase in linecount is partly explained by additional blank lines in the *ixml* version to separate different blocks of grammar constructs. For the most part, the original grammar names are retained.

There are a number of differences between ABNF and *ixml* grammars. A few of these differences are noted below.

- ABNF separates by spaces the sequences of factors (terminals and non-terminals) that make up rules, whereas *ixml* uses a comma (`','`) to separate factors in a sequence.
- An *ixml* rule ends with a period character (`'.'`) whereas ABNF rules end with a pair of CRLF chars.
- *ixml* rule names can be separated from the rule definitions with either the equals sign (`'='`) or the colon (`':'`) character whereas ABNF is always an equals sign (`'='`). *ajp* uses the equals sign for similarity to ABNF.
- ABNF alternatives within a rule are separated by a forward slash (aka *solidus*) character (`'/'`) while *ixml* alternatives can be separated by either semicolon (`';'`) or vertical bar (`'|'`) characters. *ajp* uses the vertical bar (`'|'`) character.
- *ixml* uses notations to determine the serialization of the processed input. For instance, an *ixml* terminal or non-terminal can indicate that it must be recognized as input but without output serialization by adding a minus character (`'-'`) before the rule name or the terminal or non-terminal factor.

There are many other differences between the two grammar definitions but these are among the most striking. The serialization options allow *ajp* to simplify the XSLT processing rules in certain cases. A highlighted comparison of six JSONPATH grammar rules in ABNF and ixml can be found in the following figure:

Figure 12. Comparison of a few JSONPATH rules in ixml and ABNF


RFC9535 ABNF grammar versus ixml grammar

```
jsonpath-query      = root-identifier , segments .
-root-identifier    = - "$" .
segments            = ( S , segment )* .
segment             = child-segment | descendant-segment .
child-segment       = bracketed-selection                                  |
                      ( - "."  , ( wildcard-selector   | member-name-shorthand ) ) .
descendant-segment  =   - ".." , ( bracketed-selection |
                                   wildcard-selector   |
                                   member-name-shorthand ) .
```

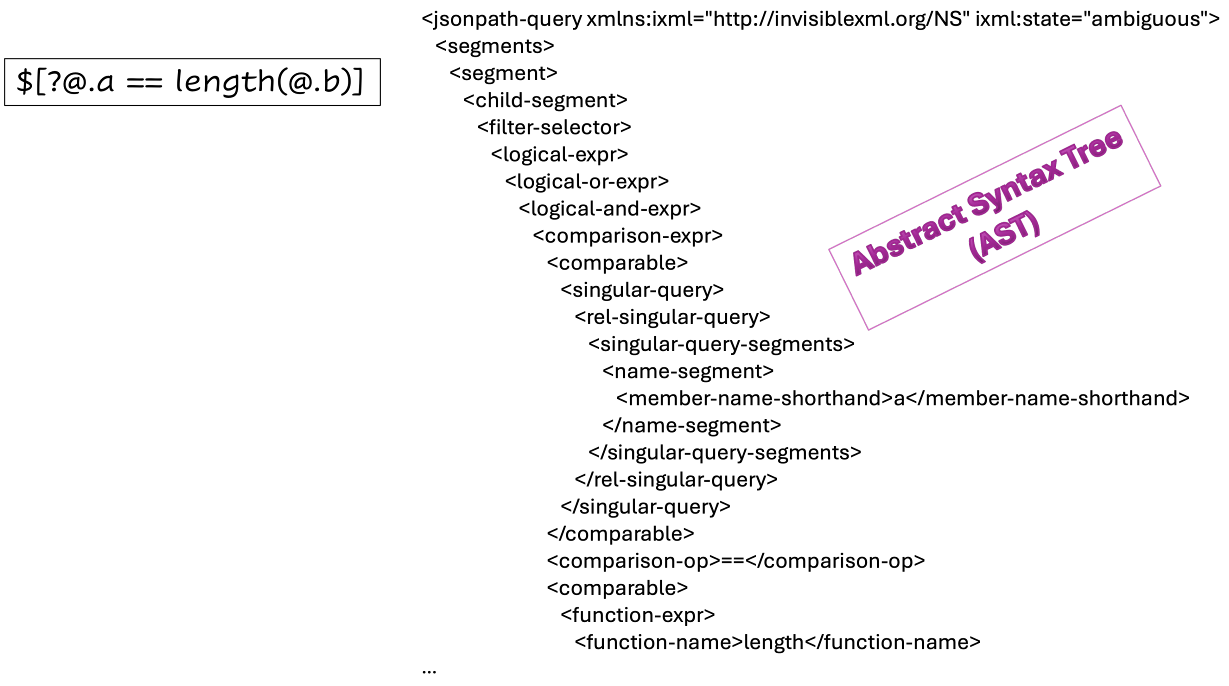
```
jsonpath-query      = root-identifier segments
root-identifier     = "$"
segments            = *(S segment)
segment             = child-segment / descendant-segment
child-segment       = bracketed-selection /
                      ("." (wildcard-selector / member-name-shorthand))
descendant-segment  = ".." (bracketed-selection /
                            wildcard-selector /
                            member-name-shorthand)
```

Amongst the differences, the terminal `"$"` and the non-termal `root-identifier` are preceeded by a `'-'` char in the *ixml* grammar to indicate that those values are not to be serialized.

7.1.2. Generating the Abstract Syntax Tree (AST) using *ixml*

The JSONPATH *ixml* grammar is loaded into the *ixml* processor to create a *parser* for the JSONPATH query. This *parser* is then invoked with the JSONPATH query to generate an XML document that constitutes the AST. The figure below gives an example of a JSONPATH query and the first part of the resultant AST.

Figure 13. A JSONPATH expression and the start of the resultant AST



```
$[?@.a == length(@.b)]
```

```
<jsonpath-query xmlns:ixml="http://invisiblexml.org/NS" ixml:state="ambiguous">
  <segments>
    <segment>
      <child-segment>
        <filter-selector>
          <logical-expr>
            <logical-or-expr>
              <logical-and-expr>
                <comparison-expr>
                  <comparable>
                    <singular-query>
                      <rel-singular-query>
                        <singular-query-segments>
                          <name-segment>
                            <member-name-shorthand>a</member-name-shorthand>
                          </name-segment>
                        </singular-query-segments>
                      </rel-singular-query>
                    </singular-query>
                  </comparable>
                  <comparison-op>==</comparison-op>
                  <comparable>
                    <function-expr>
                      <function-name>length</function-name>
...
```

7.2. An overview of `ajp:getSegments()`

A high-level overview of the *compile time* process can be shown in a few fragments of XSLT code:

Figure 14. `ajp:getSegments()`

```
<xsl:variable name="ajp:parser" as="function(*)" static="yes”
              select="cs:load-grammar('jsonpath.ixml', map { })" />

<xsl:function name="ajp:getAST" as="document-node(element(jsonpath-query))" >
    <xsl:param name="jsonpathQuery" as="xs:string" />

    <xsl:sequence select="$ajp:parser($jsonpathQuery)" />
</xsl:function>

<xsl:function name="ajp:getSegments" as="map( xs:string, array(function(*))+ )*" >
    <xsl:param name="jsonpathQuery" as="xs:string" />

    <xsl:apply-templates select="ajp:getAST($jsonpathQuery)" />
</xsl:function>
```

The variable `ajp:parser` is created at the time of the XSLT stylesheet compilation thanks to the `static` attribute on the variable element. This is the parser created by the Coffee Sacks function `cs:load-grammar()`, part of the *nineml* suite of tools. [NINEML] The parser is invoked by the function `ajp:getAST()` which passes to the parser the string containing the JSONPATH query and receives the XML document that represents the Abstract Syntax Tree in return. The function `ajp:getSegments()` calls `ajp:getAST()` and invokes `xsl:apply-templates` on the resultant AST. That's the high-level process for generating the *segments* description.

7.3. Applying templates to the AST

Looking at the templates used to create the *segments* description, we can start at the top-most templates and work our way down to the most detailed.

7.3.1. Top-level templates

The top-level templates do nothing more than recursive calls to `xsl:apply-templates`. The templates for the `segments` element produces a single map.

Figure 15. Top level templates

```
<xsl:template match="/"        as="map( xs:string, array(function(*))+ )*" >
    <xsl:apply-templates />
</xsl:template>

<xsl:template match="jsonpath-query"
                               as="map( xs:string, array(function(*))+ )*" >
    <xsl:apply-templates />
</xsl:template>

<xsl:template match="segments" as="map( xs:string, array(function(*))+ )*" >
    <xsl:apply-templates />
</xsl:template>

<xsl:template match="segment"  as="map( xs:string, array(function(*))+ )" >
    <xsl:apply-templates />
</xsl:template>
```

7.3.2. Templates that create a map for a segment

There are three templates that create a singleton map that corresponds to a *segment*. The key for the map is set to either `'child'` or `'descendant'`. The map values, a non-empty sequence of arrays of functions, are produced by recursive calls to `xsl:apply-templates`.

Figure 16. Templates that create a map for a segment

```
<xsl:template match="child-segment" as="map( xs:string, array(function(*))+ )" >
    <xsl:map-entry key="'child'" >
        <xsl:apply-templates />
    </xsl:map-entry>
</xsl:template>

<xsl:template match="descendant-segment"
                                    as="map( xs:string, array(function(*))+ )" >
    <xsl:map-entry key="'descendant'" >
        <xsl:apply-templates />
    </xsl:map-entry>
</xsl:template>

<xsl:template match="index-segment | name-segment"
                                    as="map( xs:string, array(function(*))+ )" >
    <xsl:map-entry key="'child'" >
        <xsl:apply-templates />
    </xsl:map-entry>
</xsl:template>
```

7.3.3. Templates creating an array of two functions for selectors

There are six templates that create an array of two functions — one template for each of the 5 different *selector*s and a 6th template for the `member-name-shorthand` which is a variant of the `name-selector`. There are a few things to notice about these six templates:

- As mentioned previously, the first function in the array is a *Keys()* function and the second function is a *Test()* function.
- The first five templates have the same *Test()* function — `ajp:wildcardSelectorTest#3`. This function simply returns the `true` value, meaning that the *Keys()* function is wholly determinant for inclusion of a given output node for those first five *selector*s. Only the template for the `filter-selector` has a different *Test()* function, one that is created by recursive calls to `xsl:apply-templates`.
- Four of the *Keys()* functions are created by *Partial Function Application (PFA)*. PFA is used as a means of "capturing" the compile-time information for use at runtime. This will be described in greater detail below.

This completes the list of templates for *segments* and *selectors*. The remaining XSLT templates concern the creation of functions for the `filter-selector` which is an arbitrarily complex expression.

Figure 17. Templates that create an array of functions for a selector

```
<xsl:template match="wildcard-selector"     as="array(function(*))" >
    <xsl:sequence select="[ ajp:wildcardSelectorKeys#1,
                            ajp:wildcardSelectorTest#3 ]" />
</xsl:template>

<xsl:template match="member-name-shorthand" as="array(function(*))" >
    <xsl:sequence select="[ ajp:nameSelectorKeys(?, .),
                            ajp:wildcardSelectorTest#3 ]" />
</xsl:template>

<xsl:template match="name-selector"         as="array(function(*))" >
    <xsl:variable name="key" as="xs:string" select="ajp:expand(string-literal)"/>

    <xsl:sequence select="[ ajp:nameSelectorKeys(?, $key),
                            ajp:wildcardSelectorTest#3 ]" />
</xsl:template>

<xsl:template match="index-selector"        as="array(function(*))" >
    <xsl:sequence select="[ ajp:indexSelectorKeys(?, ajp:integer(.)),
                            ajp:wildcardSelectorTest#3 ]" />
</xsl:template>

<xsl:template match="slice-selector"        as="array(function(*))" >
    <xsl:sequence select="[ ajp:sliceSelectorKeys(?, start, end, step),
                            ajp:wildcardSelectorTest#3 ]" />
</xsl:template>

<xsl:template match="filter-selector"       as="array(function(*))" >
    <xsl:variable name="filterSelectorTest" as="function(*)" >
        <xsl:apply-templates />
    </xsl:variable>

    <xsl:sequence select="[ ajp:wildcardSelectorKeys#1,
                            $filterSelectorTest ]" />
</xsl:template>
```

7.4. Partial Function Application to generate *Keys()* functions

Four of the array functions templates use Partial Function Application (*PFA*) in order to generate the *Keys()* function that has the following signature, taking a single `item()?` as an argument.

```
function keys($item as item()?) as item()*
```

The three *Keys()* functions, before *PFA*, are the following:

```
function ajp:nameSelectorKeys($item  as item()?, $match as xs:string
                             ) as item()?
{
    if (($item instance of map(*))
          and
```

```
        ($match instance of xs:string))
    then $match[ map:contains($item, $match) ]
    else ()
}

function ajp:indexSelectorKeys($item  as item()?, $match as xs:integer
                              ) as item()?
{
    if ($item instance of array(*) and $match instance of xs:integer)
    then let $length := array:size($item),
             $index := ( if ($match ge 0)
                         then ($match + 1)
                         else ($match + 1 + $length)
                       )
         return $index[(. gt 0) and (. le $length)]
         else ()
}

function ajp:sliceSelectorKeys($item  as item()?, $start as xs:integer?,
                                                  $end   as xs:integer?,
                                                  $step  as xs:integer?
                              ) as item()*
{
    if ($item instance of array(*))
    then ajp:sliceProvider($item, $start, $end, $step) => ajp:keysFromProvider()
    else ()
}
```

The *PFA* of these three *Keys()* functions is the following, from the templates:

```
[ ajp:nameSelectorKeys(?, .),                 ajp:wildcardSelectorTest#3 ]

[ ajp:indexSelectorKeys(?, ajp:integer(.)),   ajp:wildcardSelectorTest#3 ]

[ ajp:sliceSelectorKeys(?, start, end, step), ajp:wildcardSelectorTest#3 ]
```

With *PFA*, the function arguments that are not called with an *ArgumentPlaceholder* (i.e. a `'?'` character) are bound to the provided values. The signature of the resultant function (i.e. **post** *PFA*) contains only those arguments for which an *ArgumentPlacehoder* is provided. A particular utility here of *PFA* is that the information that is available at JSONPATH expression *compile time* (i.e. during the call to `ajp:getSegments()`) is "captured" by *PFA* within the resultant *partially-applied* function, ready then to be used at JSONPATH *runtime*. Also, the resultant function after *PFA* matches the signature that is required for the *Keys()* function, taking only the *runtime* argument `item()?` that comes from the JSONPATH query argument.

PFA is also used for the *Test()* functions which will be described later.

7.5. The Slice Provider: inspired by generators

The JSONPATH `slice-selector` allows for selecting values in a subset of the elements of a JSON array. The fixed parameters for the slice are: `start index`, `end index` and `step`, all of which can be absent and even negative. In the case of a negative step, the values are selected in descending index order.

The `slice-selector` specifics are provided in RFC9535 Section 2.3.4 Array Slice Selector. The *ajp* implementation of the *Keys()* function for the `slice-selector` is inspired by the *generators* proposal made by Dimitre Novatchev [GENXPATH]. Because the implementation in *ajp* is not nearly as general as the proposed *generators*, it is called here a *provider* to avoid any confusion with full-blown *generators*.

The slice keys are created by calling a series of functions that are members of the `map(*)` that makes up the `sliceProvider`. The call to `$provider?get-current()` retrieves the current value of the provider and the call to `$provider?move-next($provider)` returns an updated map containing a new *current* value, up to the point that the `end` bound has been reached.

```
function ajp:sliceSelectorKeys($item  as item()?, $start as xs:integer?,
                                                  $end   as xs:integer?,
                                                  $step  as xs:integer?
                              ) as item()*
{
    if ($item instance of array(*))
    then ajp:sliceProvider($item, $start, $end, $step) => ajp:keysFromProvider()
    else ()
}

function ajp:keysFromProvider($provider as map(*) as="item()*
{
    if ($provider?end-reached)
    then ()
    else (
        $provider?get-current($provider),
        ajp:keysFromProvider($provider?move-next($provider))
    )
}
```

This mechanism allows for lazy evaluation of the slice indices. The XPath functions for implementing the `sliceProvider` itself can be found in the *ajp* repo file `slice-provider.xslt`.

7.6. Chaining *Test()* functions with Partial Function Application

The use of PFA for creating the *Keys()* functions has been shown in the earlier sections. PFA is also used for the *Test()* functions and, especially, for chaining them together.

To demonstrate why function chaining is necessary, we can look at part of the *ixml* grammar for a `filter-selector`.

```
filter-selector  = - "?" , S , logical-expr .

logical-expr     = logical-or-expr          .

logical-or-expr  = logical-and-expr , ( S , - "||" , S , logical-and-expr )* .

logical-and-expr = basic-expr       , ( S , - "&&" , S , basic-expr       )* .
```

Here we see that a `filter-selector` is a terminal `'?'` followed by a non-terminal `logical-expr` which, itself, is simply a `logical-or-expr`. These first two rules are handled by two simple XSLT templates, for which we've seen the first template previously. These templates simply "drill down" into the AST via recursive calls to `xsl:apply-templates`.

```
<xsl:template match="filter-selector" as="array(function(*))" >
    <xsl:variable name="filterSelectorTest"
                  as="function(item(), item()?, item()?) as xs:boolean" >
        <xsl:apply-templates />
    </xsl:variable>

    <xsl:sequence select="[ ajp:wildcardSelectorKeys#1, $filterSelectorTest ]" />
</xsl:template>

<xsl:template match="logical-expr"
              as="function(item(), item()?, item()?) as xs:boolean" >
    <xsl:apply-templates />
</xsl:template>
```

The next two non-terminals, `logical-or-expr` and `logical-and-expr`, are more interesting, because they both involve an arbitrary number of factors separated by `'||'` or `'&&'`. The XSLT templates that handle these two expressions need to handle the cases where boolean algebra is to be applied.

```
<xsl:template match="logical-or-expr"
              as="function(item(), item()?, item()?) as xs:boolean" >

    <xsl:variable name="operands"
                  as="(function(item(), item()?, item()?) as xs:boolean)*" >
        <xsl:apply-templates />
    </xsl:variable>

    <xsl:sequence select="if (count(operands) eq 1)
                          then                            $operands
                          else ajp:logicalOrExpr(?, ?, ?, $operands)" />
</xsl:template>

<xsl:template match="logical-and-expr"
              as="function(item(), item()?, item()?) as xs:boolean" >

    <xsl:variable name="operands"
                  as="(function(item(), item()?, item()?) as xs:boolean)*" >
        <xsl:apply-templates />
    </xsl:variable>

    <xsl:sequence select="if (count(operands) eq 1)
                          then                             $operands
                          else ajp:logicalAndExpr(?, ?, ?, $operands)" />
</xsl:template>
```

In the case where the count of `$operands` is 1, then no *disjunction* (i.e. "or") or *conjunction* (i.e. "and") operation is necessary and the template will simply return the single function created by recursive `xsl:apply-templates`; however, if there is more than one function in `$operands`, then the appropriate boolean operation needs to be applied on the multiple operands. In this case, the template will return the result of *PFA* on `ajp:logicalOrExpr()` or `ajp:logicalAndExpr()` where the `$operands` parameter is "captured" at JSONPATH *compile time*, much in the same fashion as for the *Keys()* functions that are the result of *PFA*.

```
function ajp:logicalOrExpr($key      as item(),
                           $item     as item()?,
                           $root     as item()?,
                           $operands as function(*)*
                          ) as xs:boolean
{
    some $operand in $operands
    satisfies $operand($key, $item, $root)
}

function ajp:logicalAndExpr($key      as item(),
                            $item     as item()?,
                            $root     as item()?,
                            $operands as function(*)*
                           ) as xs:boolean
{
    every $operand in $operands
    satisfies $operand($key, $item, $root)
}
```

This term "chaining" is used in this paper to describe, as an example, the call from the function `ajp:logicalOrExpr()` to the "captured" functions in `$operands`, each time passing the three arguments for the *runtime* calls to the *Test()* functions.

7.7. Function chaining with SubQueries / SubSegments

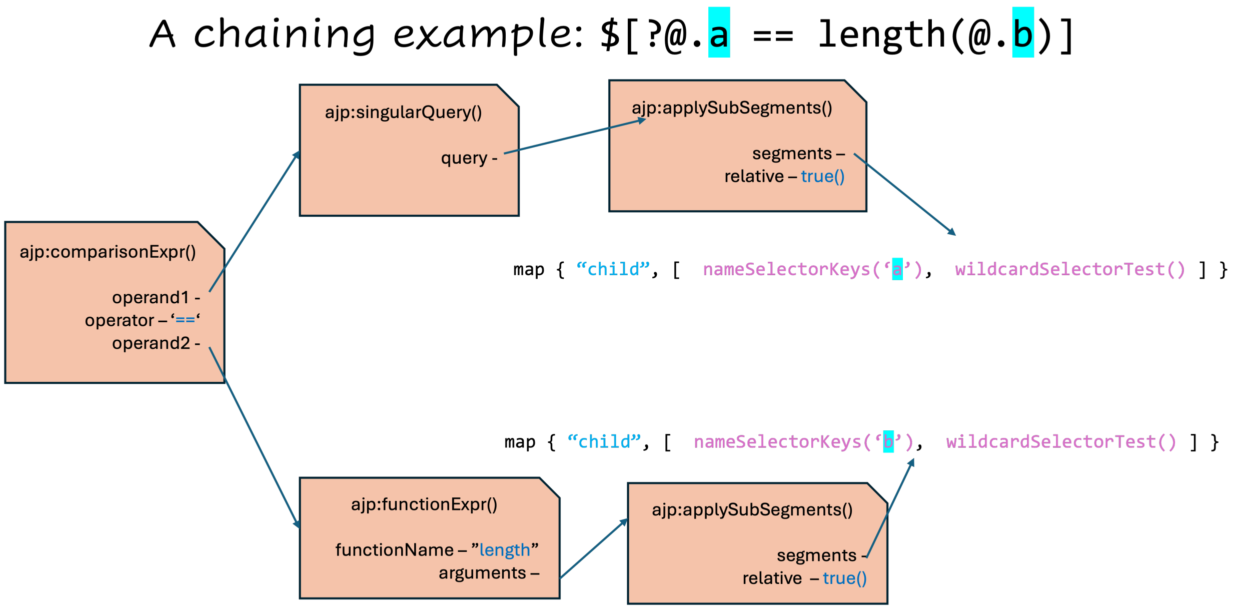
Within a `filter-selector` it is often the case that a value is referenced with a *subquery*. For example in the following JSONPATH query involving a single *child* segment having a `filter-selector`, there are two *subqueries*: `@.a` and `@.b`. It is also possible to have a subquery that references the query argument (i.e. `$root`) itself by using the `root-identifier` or `'$'`. These are examples of what is called here a *subquery*. *(N.B. RFC9535 does not employ the terms **subquery** and **subsegment** but does mention **subexpression** in section 2.3.5 Filter Selector.)*

```
$[?@.a == length(@.b)]
```

Just as in the global JSONPATH query, a *subquery* is evaluated using an input *nodelist* consisting initially of a single *node* which is either the *current node* (if the *subquery* starts with a `'@'` character) or the *root node* (if the *subquery* starts with a `'$'` character). In both cases, what follows the initial character is a possibly-empty sequence of *segments* that can be described by the same *segments* description as presented previously. To evaluate the *nodelist* results of the *subquery*, a call is made from some point in the function chain to the three-argument variant of `ajp:applySegments()` that was presented in the earlier *ajp runtime* section.

We can take the previous `filter-selector` expression as an example to view the different chained functions that result.

Figure 18. A JSONPATH expression and the start of the resultant AST



The top-level function call for the `filter-selector` will be the call to `ajp:comparisonExpr()` which will, in turn, call the functions associated with its two `comparable` operands, `ajp:singularQuery()` and `ajp:functionExpr()`. Both of these functions will chain calls to `ajp:applySubSegments()` in order to evaluate their respective *subqueries* which are represented by *segments*. The *segments* are generated at *compile time* and "captured" by *PFA* in the same fashion as described previously. The function `ajp:applySubSegments()` looks like this:

```
function ajp:applySubSegments($key      as item(),
                              $item     as item()?,
                              $root     as item()?,

                              $segments as map(xs:string, array(function(*))+)*,
                              $relative as xs:boolean
                             )          as map(xs:string, item()?           )*
{
    let $startNodelist := if ($relative)
                          then map { '@' : $item($key) }
                          else map { '$' : $root       }
    return ajp:applySegments($startNodelist, $segments, $root)
}
```

The function `ajp:applySubSegments()` is very simple: it creates the initial *nodelist* of one *node*, `$startNodeList`, using the `$relative` value "captured" by *PFA* to determine `$startNodeList`'s contents, and then calls `ajp:applySegments()` with that *nodelist*, applying the `$segments` that were "captured" by *PFA* at JSONPATH *compile time*. It is for this usage that the `$root` parameter is passed to the *Test()* functions.

7.8. JSONPATH Function Extensions

RFC9535 Section 2.4 Function Extensions describes how the `filter-selector` can call functions at certain points. These function extensions allow additional "*filter expression functionality*". Five base function extensions are defined within RFC9535 that are required in a conformant implementation: `length()`, `count()`, `match()`, `search()`, `value()`.

A type system is defined in RFC9535 Section 2.4.1 "*Type System for Function Expressions*" in order to describe both the return types of the function extensions and also the argument types of those functions. This type system is very simple but it does introduce a few new values over and above those in JSON. Reproducing here the "*Table 13: Function Extension Type System*" from RFC9535

Table 1. RFC9535 Table 13: Function Extension Type System

Type	Instances
ValueType	JSON values or Nothing
LogicalType	LogicalTrue or LogicalFalse
NodesType	Nodelists

The XML element value `$NULL` was presented in the *ajp runtime* section of this paper. The value `$NOTHING` is implemented in a similar fashion.

```
<xsl:variable name="NOTHING"      as="element()" select="ajp:element('Nothing')"     />
<xsl:variable name="NULL"         as="element()" select="ajp:element('Null')"        />

<xsl:variable name="VALUE_TYPE"   as="element()" select="ajp:element('ValueType')"   />
<xsl:variable name="LOGICAL_TYPE" as="element()" select="ajp:element('LogicalType')" />
<xsl:variable name="NODES_TYPE"   as="element()" select="ajp:element('NodesType')"   />

<xsl:function name="ajp:element" as="element()" >
    <xsl:param name="name" as="xs:string" />

    <xsl:element name="{$name}">
        <xsl:value-of select="$name" />
    </xsl:element>
</xsl:function>
```

All of these specific types and values, that is the three Function Extension types and the two specific values, `$NULL` and `$NOTHING`, are bound as *global* variables referencing unique elements for use throughout *ajp*. The function extensions for *ajp* are defined via these references:

```
<!-- N.B. Some minor changes made from original code to fit in 80 columns -->
<xsl:variable name="extensionFunctions" as="map(xs:string, map(*))"
              select="
   map:merge((
     ajp:extFunc('length', ajp:extFuncLength#1, $VALUE_TYPE,   [ $VALUE_TYPE ]),
     ajp:extFunc('count',  ajp:extFuncCount#1,  $VALUE_TYPE,   [ $NODES_TYPE ]),
     ajp:extFunc('match',  ajp:extFuncMatch#2,  $LOGICAL_TYPE, [ $VALUE_TYPE,
                                                                 $VALUE_TYPE ]),
     ajp:extFunc('search', ajp:extFuncSearch#2, $LOGICAL_TYPE, [ $VALUE_TYPE,
                                                                 $VALUE_TYPE ]),
     ajp:extFunc('value',  ajp:extFuncValue#1,  $VALUE_TYPE,   [ $NODES_TYPE ])
   ))"
/>
```

```
<xsl:function name="ajp:extFunc" as="map(xs:string, map(*))" >
    <xsl:param name="name"       as="xs:string"        />
    <xsl:param name="function"   as="function(*)"      />
    <xsl:param name="returnType" as="element()"        />
    <xsl:param name="paramTypes" as="array(element())" />

    <xsl:map-entry key="$name">
        <xsl:map>
            <xsl:map-entry key="'name'"       select="$name"       />
            <xsl:map-entry key="'function'"   select="$function"   />
            <xsl:map-entry key="'returnType'" select="$returnType" />
            <xsl:map-entry key="'paramTypes'" select="$paramTypes" />
        </xsl:map>
    </xsl:map-entry>
</xsl:function>
```

Here the required function extensions are presented with their types. Note that *ajp* contains additional function extensions that are not presented here for reasons of brevity.

At *compile time*, the function extension's parameters and result types are checked and can result in *compile time* errors. (**N.B.** RFC9535 explicitly forbids any errors at *runtime*, hence any errors are necessarily raised at *compile time*.)

As an example, when the supplied actual parameter to a function is a `filter-query` and, concomitantly, the function argument's type is defined as `$VALUE_TYPE`, an additional compile time check is required to verify that the `filter-query` is a `singular-query`. The implementation of the template that handles a `function-argument` which is a `filter-query` has this look:

```
<xsl:template match="function-argument[filter-query]" as="function(*)" >
  <xsl:param name="functionName" as="xs:string" />

  <xsl:variable name="argumentType" as="element()"
                select="ajp:argumentAtPosition($functionName, position())" />

  <xsl:variable name="query" as="function(*)" >
    <xsl:apply-templates select="filter-query"/>
  </xsl:variable>

  <xsl:sequence select="if      ( $argumentType is $NODES_TYPE )
                        then                             $query
                        else if ( $argumentType is $LOGICAL_TYPE )
                        then ajp:nodesToLogical(?, ?, ?, $query)
                        else if ( $argumentType is $VALUE_TYPE
                                   and
                                  ajp:isSingularQuery(filter-query) )
                        then ajp:singularQuery (?, ?, ?, $query)
                        else    (: $argumentType is $VALUE_TYPE !singular:)
                             ajp:error('FCT', 8, 'argument ' || position() ||
                                                 'of function ' ||
                                                 $functionName   || '()' ||
                                                 'must be a singular query.')
                       " />
```

```
25 </xsl:template>

   <xsl:function name="ajp:isSingularQuery" as="xs:boolean" >
     <xsl:param name="filterQuery" as="element(filter-query)" />

30   <xsl:sequence select="every $segment in $filterQuery//segment/*
                         satisfies name($segment) eq 'child-segment'
                                    and
                                   count($segment/*) eq 1
                                    and
35                                 exists($segment[member-name-shorthand
                                                   |
                                                   index-selector
                                                   |
                                                   name-selector])" />
40 </xsl:function>
```

This template definition may require some explanation. **Line 1** shows that the *template* will match a `function-argument` that is materialized by a `filter-query` (as opposed to being materialized by one of the other three possibilities, **i.e.** a `literal`, a `logical-expr` or a `function-expr`). **Line 5** will retrieve the `$argumentType` for this argument of this function. **Line 7** binds the *Test()* function, created by recursive `xsl:apply-templates` and that implements the `filter-query`, to a variable, `$query`. **Line 11** starts a series of chained `if` .. `then` .. `else if` statements that handle the cases presented by different function argument types. If the function argument's type is `$NODES_TYPE` then the result of the function `$query` is already in the right form, as a *Nodelist*, hence no adaptation is required and the template will simply produce the `$query` function. If the function argument's type is `$LOGICAL_TYPE`, then an adaptation is required by "chaining" via *PFA* of `ajp:nodesToLogical()`, a function which returns the value `true()`if there is at least one *Node* in the *Nodelist* returned from the "chained" call to `$query`.

At **Line 15** is the case of the function argument which is `$VALUE_TYPE` but also requires that the `filter-query` be a *singular query*, checked via the XPath function `ajp:isSingularQuery()`. (It may be interesting to note in passing that this is a very useful property of the XSLT template that the function associated with the `filter-query` has already been bound to the variable `$query` by recursive `xsl:apply-templates` but that we also have `filter-query` available for inspection by XPath.)

At **Lines 15-17**, if the `if` condition is evaluated as `true()`, then "chaining" via *PFA* is performed on the function `ajp:singularQuery()`, which returns the value of the single *Node* returned in the *Nodelist*, if there is one and, otherwise, returns the constant `$NOTHING`. If the `if` condition is evaluated as `false()`, then the unconditional `else` statement raises an XPath error via `ajp:error()` to indicate that the `filter-query` is not singular and, this, in order to better orient the query writer to a solution for the error.

7.8.1. A note on the singular query requirement in RFC9535 for VALUE_TYPE function arguments

RFC9535 Section 2.4.3 *"Well-Typedness of Function Expressions"* covers the requirements for function argument typing. During the initial implementation of *ajp*, the author of this paper did not realize that the singular query requirement, which is not constrained by the grammar, should give rise to an error.

The author's realization of the requirement was finally borne out by tests in the Compliance Test Suite, those with "non-singular" in the title. [JPTHCTS] On further searching, some mention was found in the example queries under Section 2.4.9 *"Examples"*. The author believes that this requirement could have been more explicitly stated in section 2.4.3.

7.9. Use by *ajp* of XML element types within *Nodelists* and *Segments*

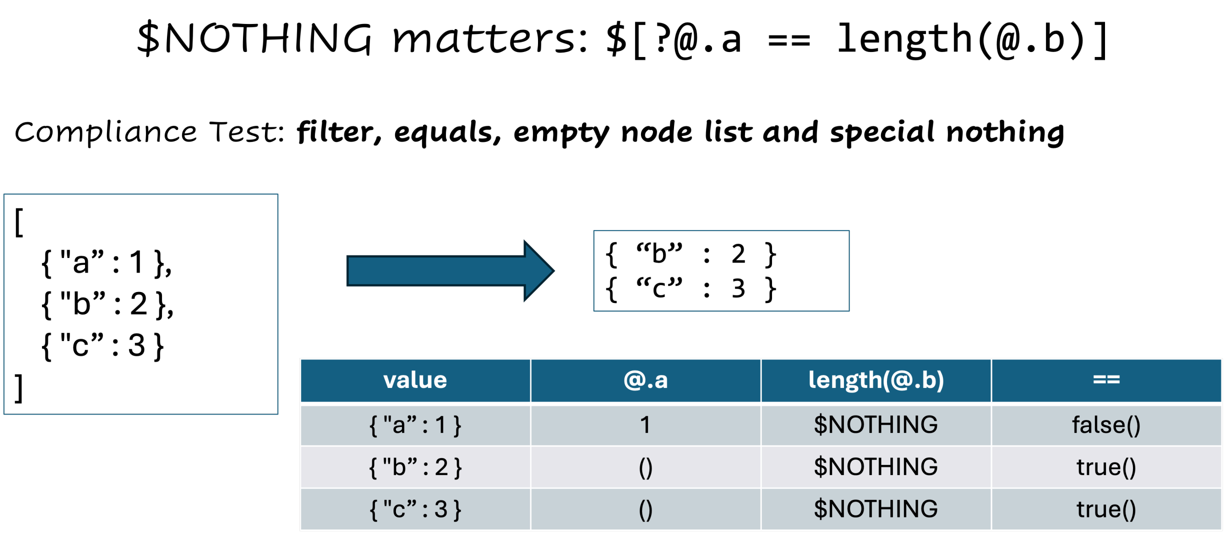
Whereas JSONPATH takes JSON values as input and output, RFC9535 mentions a value, `Nothing`, can be returned by function extensions that return values of `$VALUE_TYPE` (e.g. the function extensions `length()` and `value()`) and also by singular query expressions that produce an empty *Nodelist*,in Section 2.4.5 *"Well-Typedness of Function Expressions* Point 2 (singular query).

7.9.1. `$NOTHING` matters

Because *ajp* is based upon XPath and hence the XPath Data Model [XDM31], it is convenient to use non-JSON data, in this case an XML element, in order to distinguish this special value, `$NOTHING`.

To give an example of why `$NOTHING` matters, we can take the following JSONPATH query and its Query Argument and Resulting Nodelist.

Figure 19. `$NOTHING` matters



$NOTHING matters: `$[?@.a == length(@.b)]`

Compliance Test: **filter, equals, empty node list and special nothing**

```
[
  { "a” : 1 },
  { "b” : 2 },
  { "c” : 3 }
]
```

```
{ “b” : 2 }
{ “c” : 3 }
```

value	@.a	length(@.b)	==
{ "a” : 1 }	1	$NOTHING	false()
{ "b” : 2 }	()	$NOTHING	true()
{ "c” : 3 }	()	$NOTHING	true()

The table shows the intermediate values of the comparison, including the values produced by the *subquery* `@.a`, which is `1` for the first child object and the empty sequence `()` for the second and third children, and the value produced by the *subexpression* `length(@.b)` which, for the three child objects of the query argument, produces for each object the value `$NOTHING`. The perhaps nonintuitive *Nodelist* result, containing the second and third children, is due to the rule for equality comparison:

> *2.3.5.2.2. Comparisons*
>
> When either side of a comparison results in an empty *nodelist* or the special result `Nothing` (see Section 2.4.1):
>
> - A comparison using the operator == yields true if and only the other side also results in an empty nodelist or the special result `Nothing`.

7.9.2. `$NULL` is different from empty

RFC9535 mentions that the JSON literal `null` should not be confused with the absence of a value. In section 2.4.1 *"Type System for Function Expressions"* under *Notes:*

> *The special result `Nothing` represents the absence of a JSON value and is distinct from any JSON value, including `null`.*

Similarly, in the Section 2.6. *"Semantics of null"*

> *Note: JSON null is treated the same as any other JSON value, i.e., it is not taken to mean "undefined" or "missing".*

To convert JSON text serializations into XPath Data Model values, the XPath function `fn:parse-json()`, under Rule 6, *"[t]he JSON value `null` is converted to the empty sequence."*

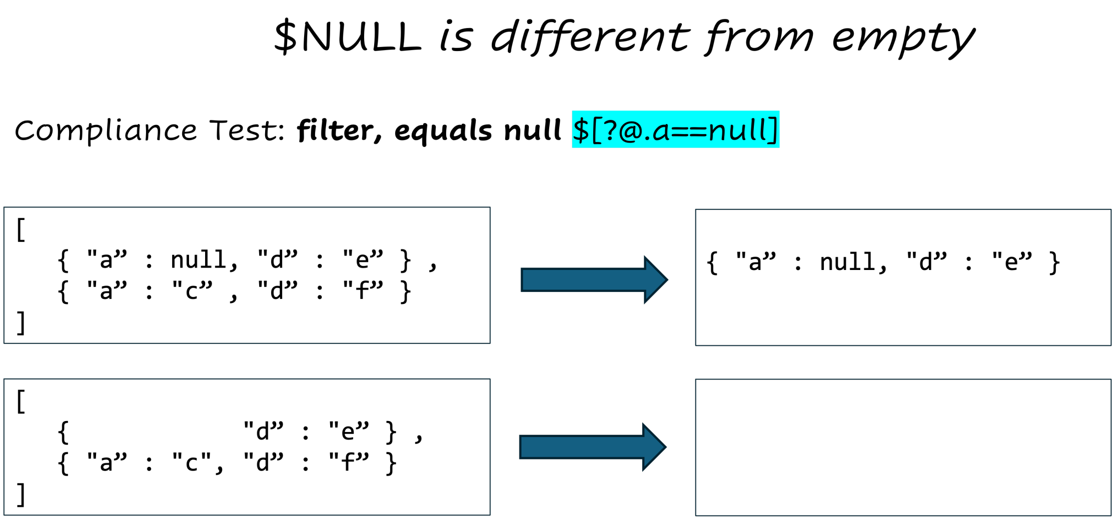
Because the empty sequence is used as the result of many different operations. Take, for instance, the return value for

```
$a('b')
```

where `$a` is a map. This expression will return the empty sequence for both the situation where the map $a has no entry with the key 'b' and the situation where `$a` has an entry with the key `'b'` and value of empty sequence. In order to disambiguate these result values, it is convenient to use the XML element value `$NULL` for the results that reference an isolated JSON literal `null` value.

Any JSONPATH queries that compare a value to JSON literal `null` will rely on this disambiguation. An example of such a test is below.

Figure 20. `$NULL` is different from empty



$NULL *is different from empty*

Compliance Test: **filter, equals null** $[?@.a==null]

```
[
   { "a" : null, "d" : "e" } ,
   { "a" : "c" , "d" : "f" }
]
```

```
{ "a" : null, "d" : "e" }
```

```
[
   {            "d" : "e" } ,
   { "a" : "c", "d" : "f" }
]
```

It is for this reason that the two-argument version of `ajp:applySegments` converts any `$NULL` values in the final return *nodelist* to empty sequences.

8. Conclusions

ajp passes all current `JSONPATH Compliance Test Suite` tests and all "compliant" Consensus Tests from the `json-path-comparison` project.

The XPath functions that comprise the *ajp runtime evaluation* of JSONPATH query expressions, once these are compiled into *segment* structures, are simple, fairly intuitive, and close to the definitions of *segment* and *selector* processing as specified within RFC9535. The *segment* structures employ *higher-order* functions, particularly the *Keys()* and *Test()* functions that are created dynamically and stored within the *selector* arrays.

The utility and flexibility of Invisible XML (*ixml*) is confirmed by the conversion of the RFC9535 ABNF grammar for JSONPATH into an *ixml* grammar. The conversion of the Abstract Syntax Tree, created by the *ixml* processor, into the *segments* structure via XSLT shows template rules that, at all times, can be related directly to the RFC9535 specification. Partial Function Application is employed at *compile time* to "capture" the information from the JSONPATH query into the partially-applied *Keys()* and *Test()* functions that are then invoked at *runtime*.

Non-JSON (i.e. XML) data is used to describe the special value `Nothing` from RFC9535, as well as describing the JSON literal `null` value when stored in isolation (i.e. not within a JSON object or JSON array).

9. Possible future work on *ajp*

ajp remains a work in progress. There are a few avenues for future work that can be described here.

- In order to be in the list of known JSONPATH implementations, it will be good to add *ajp* to the `json-path-comparison` project. [BURGCMP]
- Integrate other *ixml* implementations, especially in order to make the implementation available in other host languages.
- Where currently there is an API available for Java, add APIs for other languages such as C, Python, Javascript, C#, etc.
- As an exercise in style, convert the XSLT part to pure XPath or possibly with XQuery.
- Because *ajp* is an XSLT 3.0 package, it requires some setup to use within an environment. It will be interesting to incorporate different packaging mechanisms into *ajp* for easier inclusion into environments.
- Package *ajp* for known environments such as eXistDB, Elemental, BaseX.

Bibliography

[RFC9535] S. Gössner, G. Normington, and C Bormann. *JSONPath: Query Expressions for JSON*. 2024. Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/rfc9535/.

[BURGCMP] Christoph Burgmer. *json-path-comparison*. https://cburgmer.github.io/json-path-comparison.

[JPTHORIG] *JSONPath - XPath for JSON*. Stefan Gössner. 2007-02-21. https://goessner.net/articles/JsonPath.

[XPATH31] Jonathan Robie, Michael Dyck, and Josh Spiegel. *XML Path Language (XPath) 3.1*. 2017-03-21. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-31/.

[XPATH31F] Dr. Michael Kay. *XPath and XQuery Functions and Operators 3.1*. 2017-03-21. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-functions-31/.

[XSLT30] Dr. Michael Kay. *XSL Transformations (XSLT) Version 3.0*. 2017-06-08. World Wide Web Consortium (W3C). https://www.w3.org/TR/xslt-30/.

[XDM31] Norman Walsh, John Snelson, and Andrew Coleman. *XQuery and XPath Data Model 3.1*. 2017-03-21. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-datamodel-31/.

[JPTHCTS] *JSONPath Compliance Test Suite*. Glyn Normington. et al. https://github.com/json-path-standard/jsonpath-compliance-test-suite.

[PRAG2024] Alan Painter. *JSONPath: an IETF Proposed Standard, with comparisons to XPath*. 2024-06-07. XML Prague. https://archive.xmlprague.cz/2024/files/xmlprague-2024-proceedings.pdf#page=47.

[IXML10] Steven Pemberton. *Invisible XML Specification*. 2022-06-20. CWI, Amsterdam. https://invisiblexml.org/1.0/.

[NINEML] Norman Tovey-Walsh. *nineml -- A family of Earley and Generalized LL (GLL) parsing tools*. https://github.com/nineml/nineml.

[RFC5234] D. Crocker and P. Overell. *Augmented BNF for Syntax Specifications: ABNF*. https://www.rfc-editor.org/rfc/rfc5234.txt.

[GENXPATH] Dimitre Novatchev. *Generators in XPath*. https://medium.com/@dimitrenovatchev/generators-in-xpath-987a609cfbd5?sk=6334d48f9565f78eba90b212e461243b.