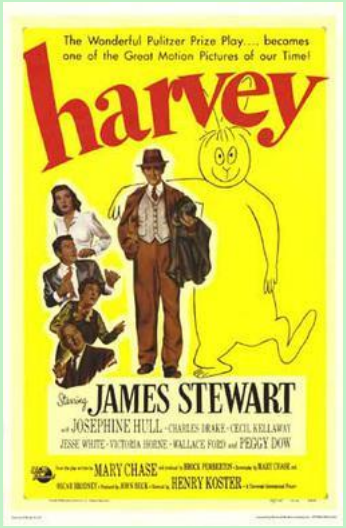
An experience in implementing JSONPATH in XPath / XSLT

with help from an Invisible Friend

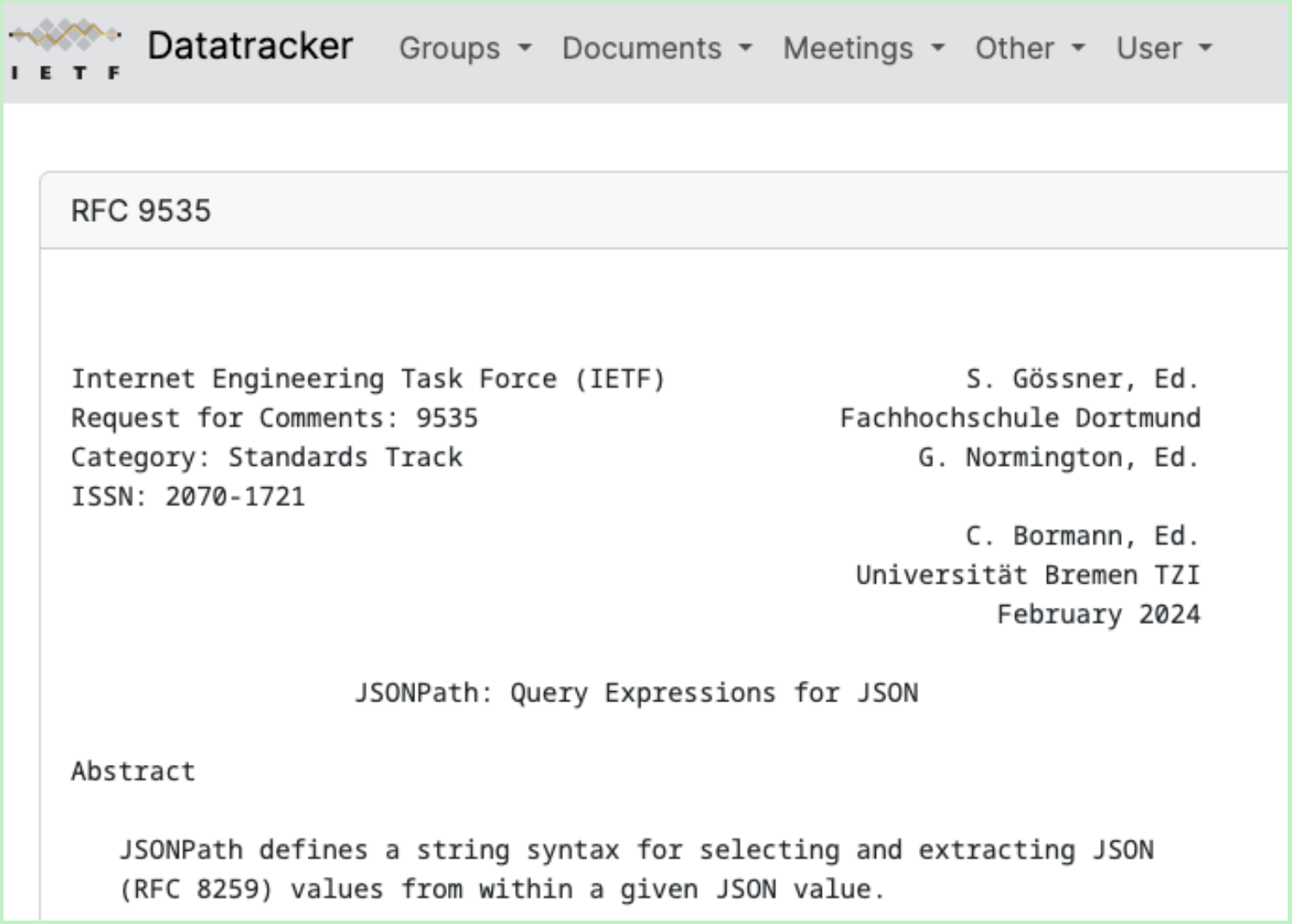


Alan Painter

Presented at Declarative Amsterdam

7 November 2025

RFC9535 – JSONPATH – appeared Feb 2024



Datatracker Groups Documents Meetings Other User

RFC 9535

Internet Engineering Task Force (IETF)
Request for Comments: 9535
Category: Standards Track
ISSN: 2070-1721

S. Gössner, Ed.
Fachhochschule Dortmund
G. Normington, Ed.

C. Bormann, Ed.
Universität Bremen TZI
February 2024

JSONPath: Query Expressions for JSON

Abstract

JSONPath defines a string syntax for selecting and extracting JSON (RFC 8259) values from within a given JSON value.

- A standard for JSONPATH after around 50 implementations were developed.

Why another JSONPATH implementation?

- Most JSONPATH implementations are pre-RFC9535 (i.e. not RFC9535).
- Some of the JSONPATH requirements can showcase XSLT/XPath features.
- The heart of RFC9535 is a context-free ABNF grammar for JSONPATH queries; **ixml** handles this really well.
- The closeness of the implementation to the grammar can mean that the implementation sheds some light on the specification.

ajp: A JSONPATH Processor: Status

Repo: https://github.com/xmljacquard/ajp/

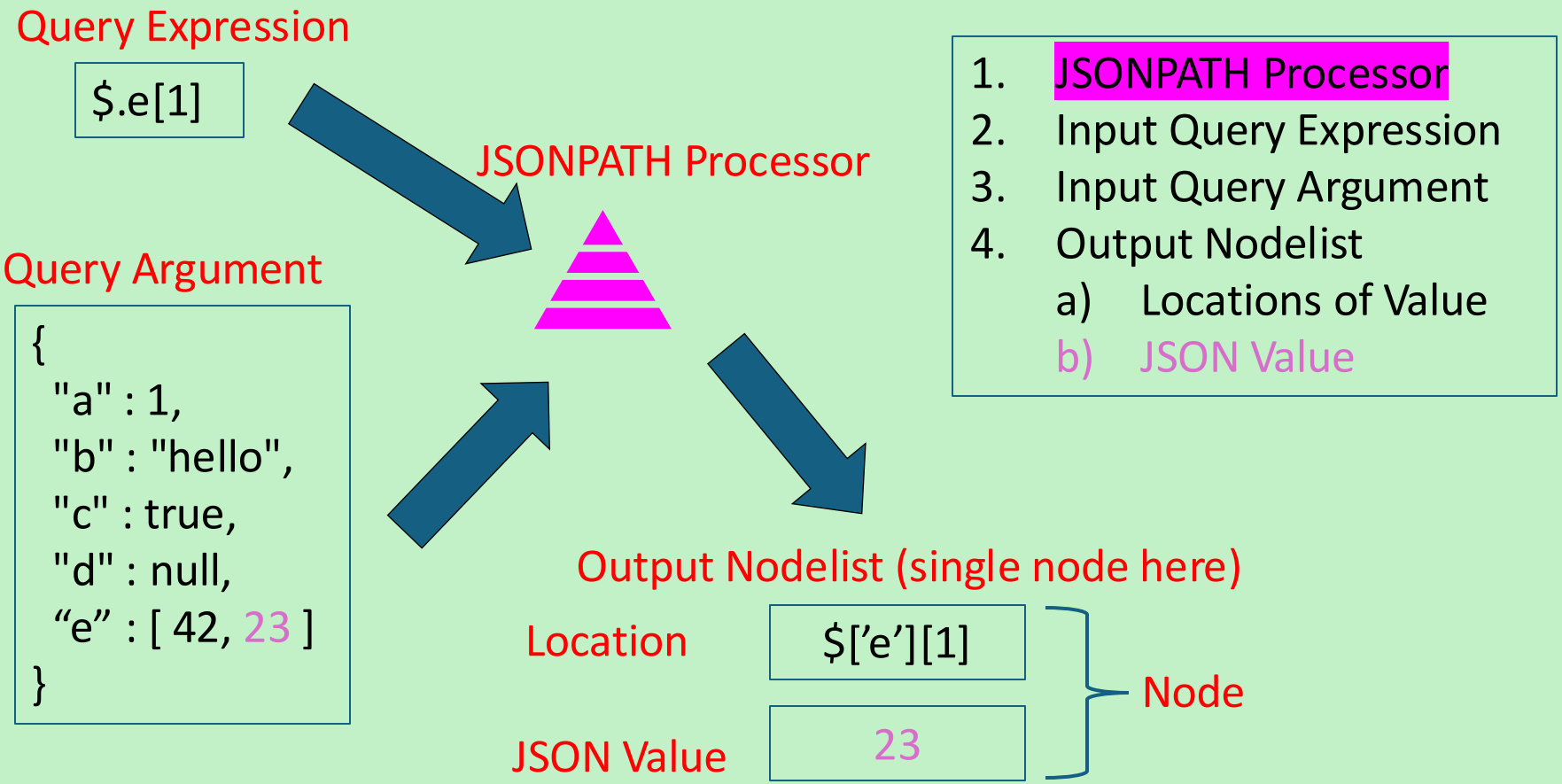
Compliance Tests Results:
https://xmljacquard.github.io/ajp/ajp-compliance-tests-report.html

Origin	Number of Tests	Number of Tests Passed	Number of Tests Failed
cts	692	692	0
ajp	17	17	0
Total	709	709	0

Consensus Tests Report:
https://xmljacquard.github.io/ajp/ajp-consensus-report.html

Result Category	Number of Tests in Category
Results Match Consensus Values in Order	148
Results Match Consensus Values in a Different Order	10
Results with a different value from the Consensus	3
Tests for which there is No Consensus	96
Total	257

A simple example of JSONPATH



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'] – descendant 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

A three-phase process for JSONPATH

Query Expression

$.e[1]

1

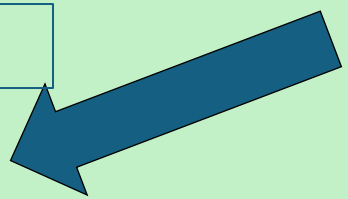
ixml

```
<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

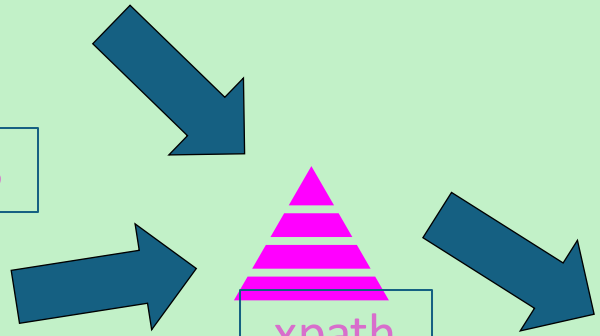
xslt



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

Query Argument

3



xpath

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

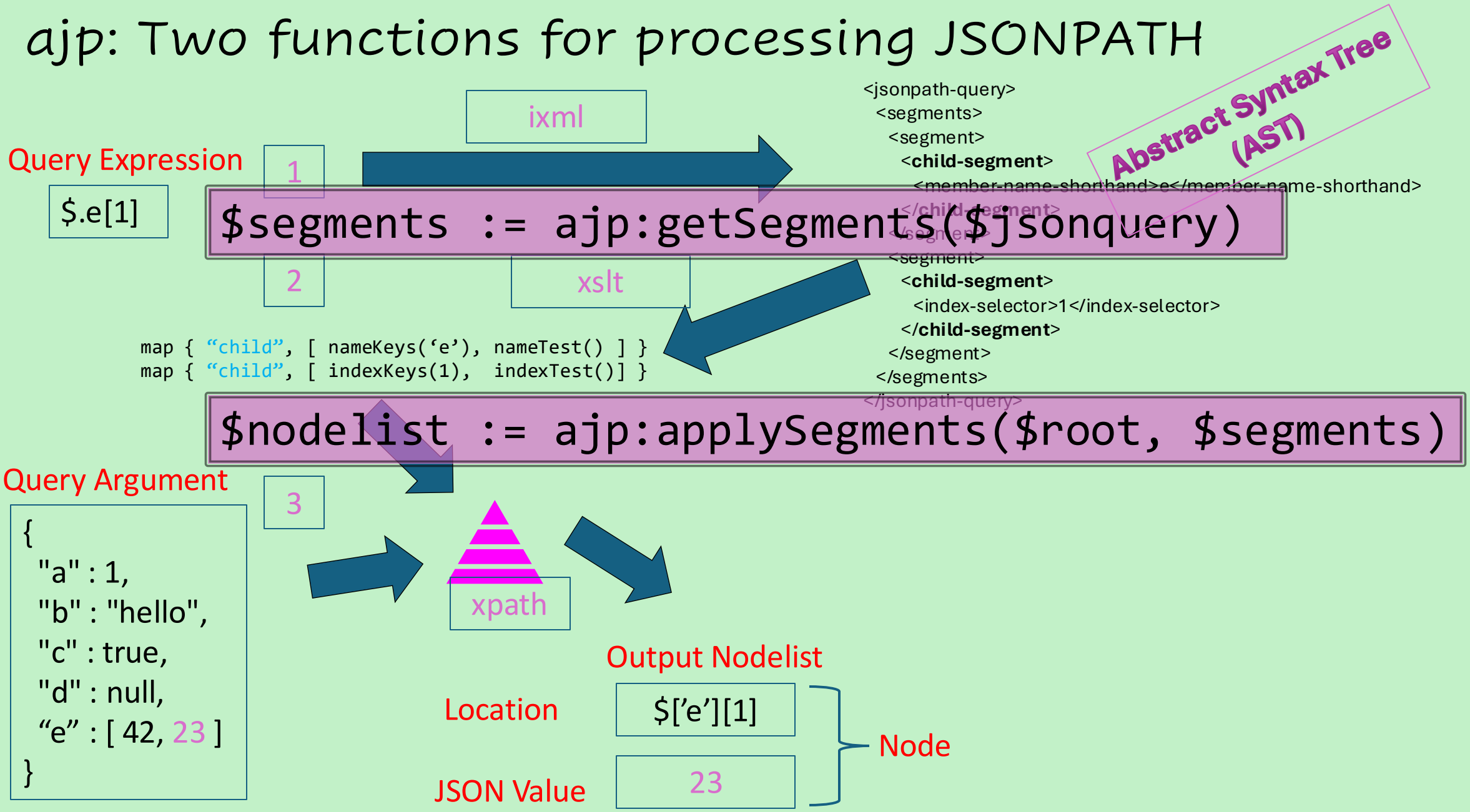
Output Nodelist

Location $['e'][1]

JSON Value 23

Node

ajp: Two functions for processing JSONPATH


Query Expression
$.e[1]
1
ixml
$segments := ajp:getSegments($jsonquery)
2
xslt
map { "child", [nameKeys('e'), nameTest()] }
map { "child", [indexKeys(1), indexTest()] }
<jsonpath-query>
<segments>

<child-segment>
<member-name-shorthand>e</member-name-shorthand>
</child-segment>

<child-segment>
<index-selector>1</index-selector>
</child-segment>

</segments>
</jsonpath-query>
Abstract Syntax Tree (AST)
$nodelist := ajp:applySegments($root, $segments)
Query Argument
3
{
"a" : 1,
"b" : "hello",
"c" : true,
"d" : null,
"e" : [42, 23]
}
xpath
Output Nodelist
Location
$['e'][1]
JSON Value
23
Node

Main Data Structures : segments and nodelists

$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() ]  }
```

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

```
map { "$['a']", 1 }
```

```
map { "$['e']", [42, 23] }
```

```
map { "$['e'][0]", 42 }
```

```
map { "$", ** }
```

The 2 selector functions in the array

[keys(), test()]

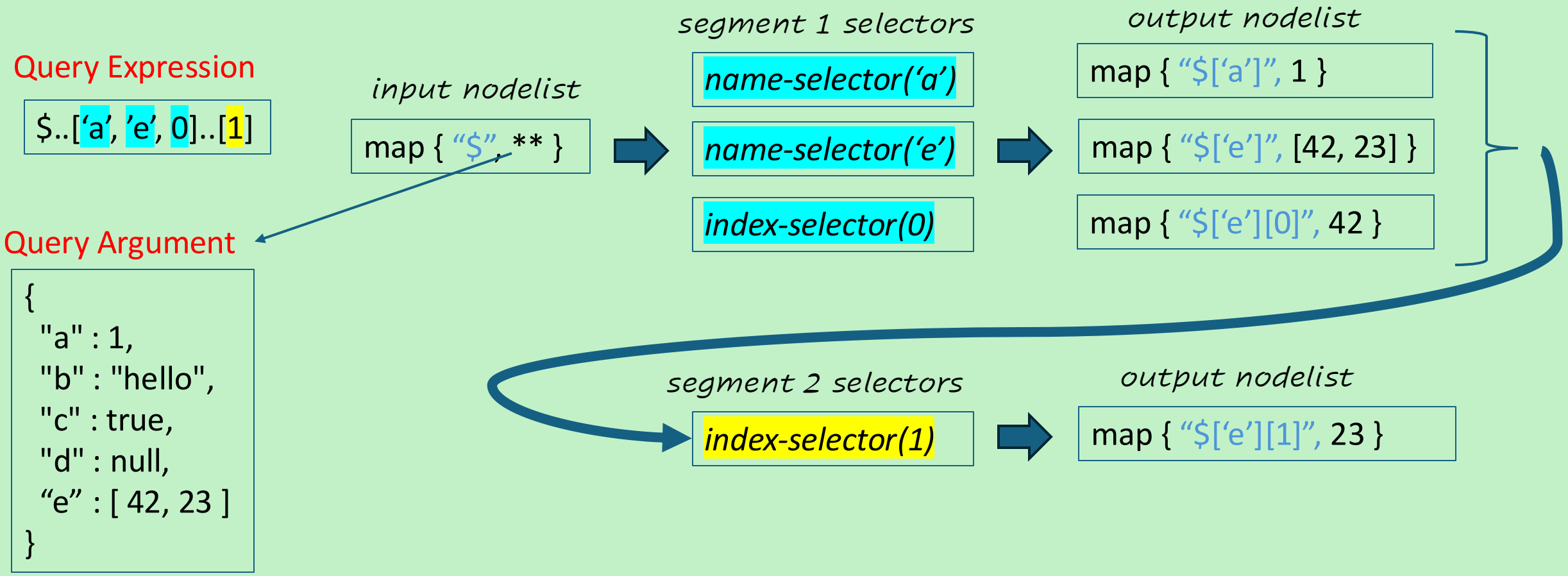
function keys($item as item()?) as item()*

- 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

Segment Processing: Nodelist Input / Output



ajp:applySegments($root, $segments)

```
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
}
```

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

```
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)
}
```

ajp:children($nodelist, $selectors, $root)

```
function ajp:children($nodelist  as map(xs:string, item()?)*,
                      $selectors as array(function(*))+,
                      $root      as item()?
                     )           as map(xs:string, item()?)* {

    for $node         in $nodelist [ ajp:isMapOrArray(.?*) ],
        $parentPath   in ajp:key($node),
        $parentValue  in $node?*,
        $selector     in $selectors,
        $key          in $selector(1)($parentValue),
        $path         in ajp:path($parentPath, $key)
    return map {
                    $path : ( $parentValue($key), $NULL )[1]
                } [ $selector(2)($key, $parentValue, $root) ]
}
```

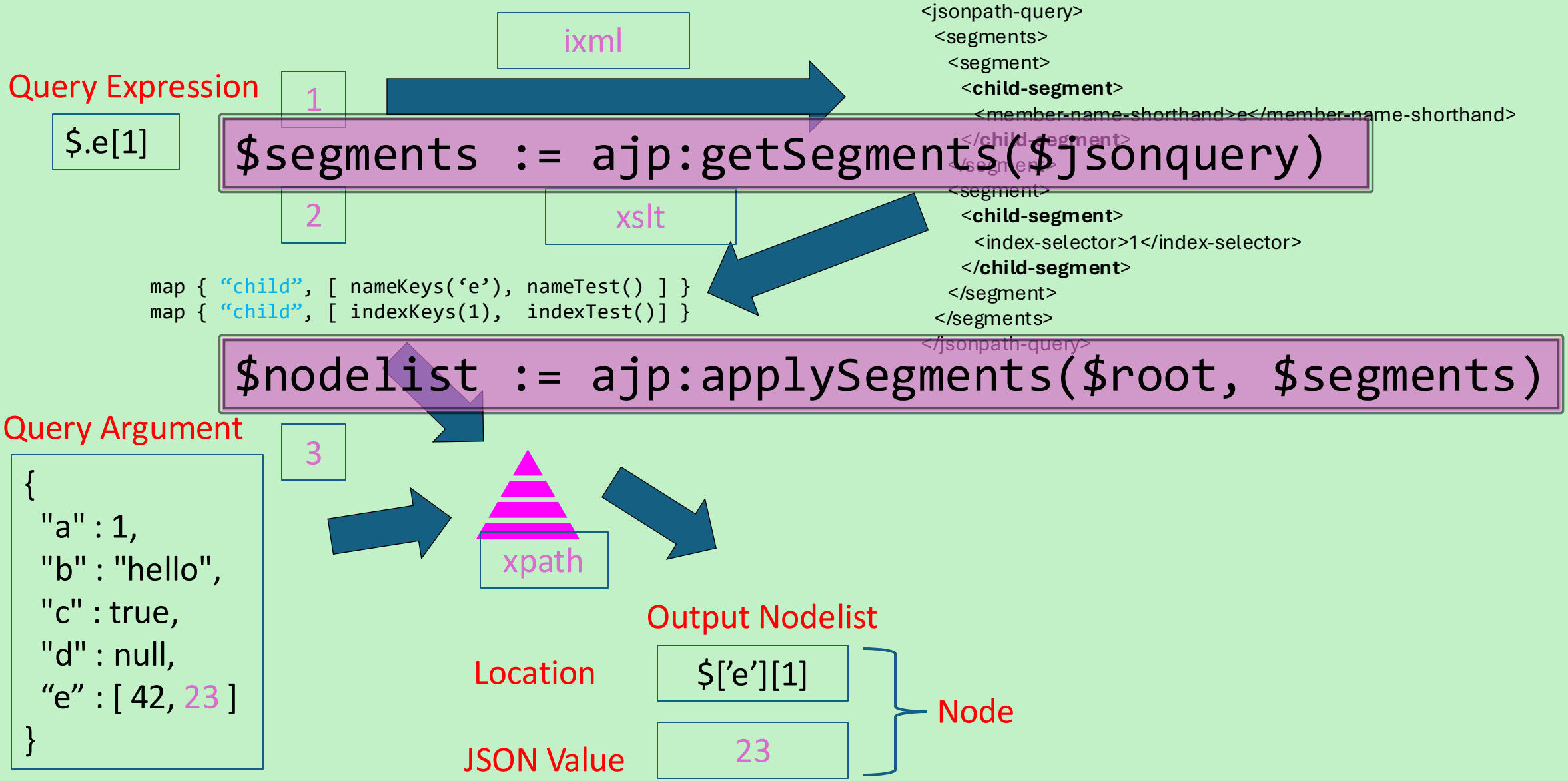
ajp:descendants($nodelist, $selectors, $root)

```
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 )
}
```

ajp:path($path, $key)

```
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 ()

}
```

ajp: Two functions for processing JSONPATH (bis)


Query Expression
$.e[1]
ixml
1
$segments := ajp:getSegments($jsonquery)
2
xslt
<jsonpath-query>
<segments>

<child-segment>
<member-name-shorthand>e</member-name-shorthand>
</child-segment>

<child-segment>
<index-selector>1</index-selector>
</child-segment>

</segments>
</jsonpath-query>
map { "child", [nameKeys('e'), nameTest()] }
map { "child", [indexKeys(1), indexTest()] }
$nodelist := ajp:applySegments($root, $segments)
Query Argument
3
{
"a" : 1,
"b" : "hello",
"c" : true,
"d" : null,
"e" : [42, 23]
}
xpath
Output Nodelist
Location
$['e'][1]
JSON Value
23
Node

ixml: using NTW's nineml (https://github.com/nineml)

```
$[?@.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>
...
```

Abstract Syntax Tree (AST)

RFC9535 ABNF grammar versus ixml grammar

ixml

```
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 ) .
```

ABNF

```
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)
```

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>
```

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>
```

Templates creating a map

```
<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>
```

Templates creating an array of 2 functions (1)

```
<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>
```

Templates creating an array of 2 functions (2)

```
<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(item(), item()?, item()?) as xs:boolean" >
        <xsl:apply-templates />
    </xsl:variable>

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

Partial function application to “capture” query data

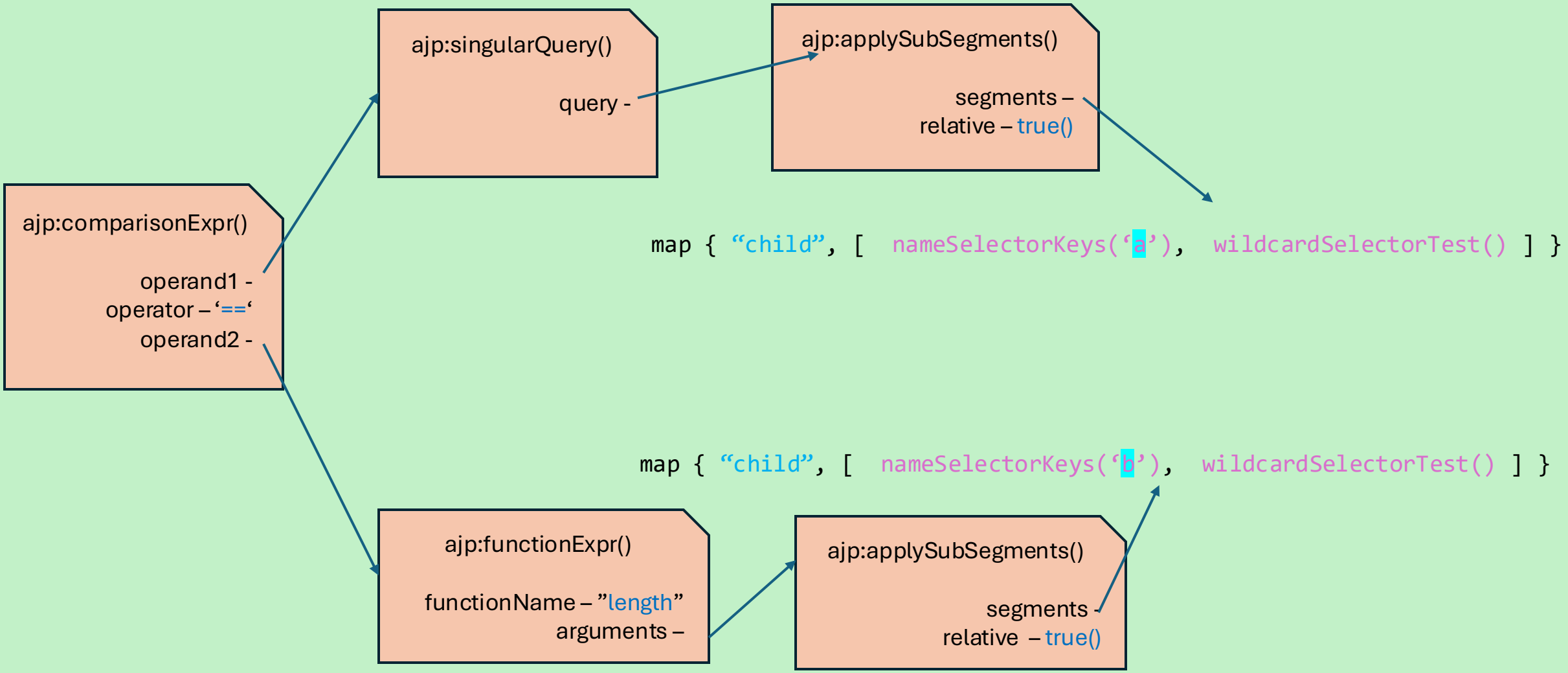
```
<xsl:function name="ajp:sliceSelectorKeys" as="item()*" >
    <xsl:param name="item"  as="item()?"     />

    <xsl:param name="start" as="xs:integer?" />
    <xsl:param name="end"   as="xs:integer?" />
    <xsl:param name="step"  as="xs:integer?" />

    <xsl:sequence select="if ($item instance of array(*))
                          then ajp:sliceProvider($item, $start, $end, $step) => ajp:keysFromProvider()
                          else ()
                         " />
</xsl:function>

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

Chaining Test functions from partial function application

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

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

    <xsl:variable name="operator" as="xs:string" select="comparison-op" />

    <xsl:sequence select="ajp:comparisonExpr(?, ?, ?, $operands[1], $operator, $operands[2])" />
</xsl:template>
```

```
comparison-expr         = comparable , S , comparison-op , S , comparable .

comparable              = literal         |
                          singular-query  | { singular query value }
                          function-expr   . { ValueType            }
```

A chaining example: $[?@.a == length(@.b)]



Applying subsegments

```
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)
}
```

A singular matter

```
<xsl:template match="function-argument[filter-query]"  … >

    <xsl:variable name="query" as="function(item(), item()?, item()?) as map(xs:string, item()?)*" >
        <xsl:apply-templates select="filter-query"/>
    </xsl:variable>

    <xsl:sequence select="if      ( $argumentType is $LOGICAL_TYPE )
                          then ajp:nodesToLogical(?, ?, ?, $query)
                          else if ( $argumentType is $VALUE_TYPE and ajp:isSingularQuery(filter-query) )
                          then ajp:singularQuery (?, ?, ?, $query)
```

```
function ajp:isSingularQuery($filterQuery as element()) as xs:boolean
{
    every $segment in $filterQuery//segment/*
    satisfies name($segment) eq 'child-segment’
               and
              count($segment/*) eq 1
               and
              exists($segment[member-name-shorthand |
                              index-selector        |
                              name-selector])
}
```

Much Ado About $NOTHING

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

<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>
```

2.4.1 Type System for Function Expressions:
ValueType : JSON values or Nothing

2.4.3. Well-Typedness of Function Expressions
. . .
- If the query results in an empty nodelist, the argument is the special result Nothing.

2.4.4 length() – returns unsigned integer or Nothing
2.4.8 value() – returns the single value argument or Nothing

$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()

2.3.5.2.2. Comparisons

* A comparison using the operator == yields true if and only the other side also results in an empty nodelist or the special result Nothing.

$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” }
]
```

ajp:singularQuery()	*if (count($nodelist) eq 0) then $NOTHING else if (count($nodelist) eq 1) then ($nodelist?*, $NULL)[1]*

Some conclusions

- ixml, xpath and xslt allow us to create an implementation that is very close to the specification.
- Template matching, higher-order functions, partial function application help in creating the selector functions, especially the filter-selector.
- Non-JSON XDM data types allow us to follow the specification for $NOTHING and handle $NULL.
- The XSLT templates can be related to the specification directly thanks to re-use of the grammar rule names in the AST.

ajp: Planned Next Steps

- Add ‘ajp’ to the ‘json-path-comparison’ repo
- Get ‘ajp’ to work with other ‘ixml’ implementations
- Whereas currently we have an API for Java, provide APIs for C, Python, Javascript, C#, perhaps others
- Have an all-XPath/XQuery version (by converting the XSLT parts from XSLT to XPath/XQuery).
- Look at constructs for facilitating the loading of XSLT packages
- Package ‘ajp’ for eXistDB, Elemental, BaseX

Contact: **contact@xmljacquard.org**

Thanks!