# A Proposal for XSLT 4.0

Michael Kay
*Saxonica*
<mike@saxonica.com>

**Abstract**

*This paper defines a set of proposed extensions to the XSLT 3.0 language [18], suitable for inclusion in version 4.0 of the language were that ever to be defined.*

*The proposed features are described in sufficient detail to enable the functionality to be understood and assessed, but not in the microscopic detail needed for the eventual language specification.*

*Brief motivation is given for each feature. The ideas have been collected by the author both from his own experience in using XSLT 3.0 to develop some sizable applications (such as an XSLT compiler: see [4], [3]), and also from feedback from users, reported either directly to Saxonica in support requests, or registered on internet forums such as StackOverflow.*

## 1. Introduction

The W3C is no longer actively developing the XSLT and XPath languages, but this does not mean that development has to stop. There is always the option of some other organisation taking the language forward; the W3C document license under which the specification is published [1] explicitly permits this, though use of the XSLT name might need to be negotiated.

This paper is a sketch of new features that could be usefully added to the language, based on experience and feedback from users of XSLT 3.0.

XSLT 3.0 (by which I include associated specifications such as XPath 3.1) introduced some major innovations [18]. A major theme was support for streaming, and by and large that aspect of the specification proved successful and complete; I have not felt any need to propose changes in that area. Another major innovation was packages (the ability to modularize a stylesheet into separate units of compilation). I suspect that there is room for polishing the spec in this area, but to date there has been relatively little feedback from users, so it is too early to know where the improvement opportunities might lie. The third major innovation concerns the data model, with the introduction of maps, arrays, JSON support, and higher-order functions, and it is in these areas that most of the proposals in this

[1]See https://www.w3.org/Consortium/Legal/2015/doc-license

paper fall, reflecting that there has been signficant user experience gained in these areas.

Some of this user experience comes from projects in which the author has been directly involved, notably:

- Development of an XSLT compiler written in XSLT, reported in [4] and [3]. The resulting compiler, at the time of publication of this paper, is almost ready for release.
- Development of an XSD validator written in XSLT, reported in [2] (The project as described was 90% completed, but the code has never been released).
- An evaluation of the suitability of XSLT 3.0 for transforming JSON files, reported at XML Prague [1].

These projects stretched the capabilities of the XSLT language and in particular involved heavy use of maps for representing data structures.

Other feedback has come from users attempting less ambitious projects, and typically reporting difficulties either directly to Saxonica or on internet forums such as StackOverflow.

The paper is concerned only with the technical content of the languages, and not with the process by which any new version of the standards might be agreed. In practice XSLT development is now being undertaken by only a small handful of implementors, and therefore a more lightweight process for agreeing language changes might be appropriate.

The proposal involves changes to the XPath language and the function library as well as to XSLT itself. In this paper, rather than organise material according to which specification is affected, I have arranged it thematically, so that the impact of related changes can be more easily assessed. I have also tried to organise it so that it can be read sequentially; I try never to use a new feature until it has been introduced.

## 2. Types

Types are fundamental to everything else, so I will start with proposed modifications to the type system.

XSLT 3.0 (by which I include XPath 3.1) enriches the type system with maps and arrays, which greatly enhances the power of the language. But experience has shown some limitations.

### 2.1. Tuple types

Maps in XSLT 3.0 are often used in practice for structures in which the keys are statically known. For example, a complex number might be represented as `map{"r": 1.0e0, "i": -1.0e0}`. Declaring the type of this construct as `map(xs:string, xs:double)` doesn't do it justice: such a type definition allows many values that don't actually represent complex numbers.

I propose instead to allow the type of these values to be expressed as `tuple(r as xs:double, i as xs:double)`.

Note that I'm not introducing tuples as a new kind of object here. The values are still maps, and the set of operations that apply to tuples are exactly the same as the operations that apply to maps. I'm only introducing a new way of describing and constraining the type.

A few details on the specification:

- The field names (here `r` and `i`) are always `xs:string` instances (for a map to be valid against the tuple type definition, the keys must match these strings under the same-key comparison rules). Normally the names must conform to the rules for an `xs:NCName`; but to allow processing of any JSON object, including objects with keys that contains special characters such as spaces, I allow the field names to be arbitrary strings; if they are not `NCName`s, they must be written in quotes.
- If the type allows the value of an entry to be empty (for example `middle` in `tuple(first as xs:string, middle as xs:string?, last as xs:string)` then the relevant entry can also be absent. Values where the entry is absent can be distinguished from those where the entry is present but empty using `map:contains()`, but both satisfy the type.
- The as clause may be omitted (for example `tuple(r, i)`). This is especially useful when tuple types are used as match patterns, where it is only necessary to give enough information to give an unambiguous match. Contrary to convention, the default type for a field is not `item()*` but rather `item()+`: this ensures that a type such as `tuple(ssn)` will only match a map if the entry with key `ssn` is actually present.
- A tuple type may be defined as extensible by adding `,*` to the list of fields, for example `tuple(first as xs:string, middle as xs:string?, last as xs:string, *)`. An extensible tuple type allows the map to contain entries additional to those listed, with no constraints on the keys or values; an inextensible tuple type does not allow extra entries to appear.
- The subtype-supertype relation is defined across tuple types in the obvious way: a tuple type `T` is a subtype of `U` if we can establish statically that all instances of `T` are valid instances of `U`. This will take into account whether `U` is extensible. Similarly a tuple type may be a subtype of a map type: for example `tuple(r as xs:double, i as xs:double)` is a subtype of `map(xs:string, xs:anyAtomicType+)`. By transitivity, a tuple is therefore also a function.
- A processor is allowed to report a static error for a lookup expression `X?N` if it can establish statically that `X` conforms to a tuple type which does not allow an

entry named `N`. For example if variable `$c` is declared with the type `tuple(r as xs:double, i as xs:double)`, then the expression `$c?j` would be a static error. (Note also that `1 to $c?i` might give a static type error, because the processor is able to infer a static type for `$c?i`)

However, a dynamic lookup in the tuple for a key that is not a known field succeeds, and returns an empty sequence. This is to ensure that tuples are substitutable for maps.

- If a variable or function argument declares its required type as a tuple type, and a map is provided as the supplied value, then the map must strictly conform with the tuple type; no coercion is performed. For example if the required type has a field declared with `i as xs:double` then the value of the relevant entry in the map must actually be an `xs:double`; an `xs:integer` will not be promoted.

## 2.2. Union Types

XSLT 3.0 and XPath 3.1 provide new opportunities for using union types. In particular, it is now possible to define a function that accepts an argument which is, for example, either an `xs:date` or `xs:dateTime`. But this can only be achieved by defining a new union type in a schema and importing the schema, which is a rather cumbersome mechanism.

I therefore propose to allow anonymous union types to be defined inline: for example `<xsl:param name="arg" as="union(xs:date, xs:dateTime, xs:time)"/>`. The semantics are exactly the same as if the same union type were defined in a schema.

The member types must be *generalized atomic types* (that is, atomic types or simple unions of atomic types), which means that the union is itself a generalized atomic type.

## 2.3. Node types

The `element()` and `attribute()` node types are extended to allow the full range of wildcards permitted in path expressions: for example `element(*:local)`, `attribute(xml:*)`. This is partly just for orthogonality (there is no reason why node types and node tests should not be 100% aligned, and this is one of the few differences), and partly because it is actually useful, for example, to declare that a template rule returns elements in a particular namespace.

This means that patterns such as `match="element(xyz:*, xs:date)` become possible, matching all elements of type `xs:date` in a particular namespace. The default priorities for such patterns are established intuitively: the priority when `foo:*` or `*:bar` is used is midway between the priorities for a full name like `foo:bar`, and the generic wildcard `*`. Since `element(*, T)` has priority 0, while `element(N, T)` is 0.25, this means the priority for `element(p:*, T)` is set at 0.125.

## 2.4. Default namespace for types

The XPath static context defines a default namespace for elements and types. I propose to change this to allow the default namespace for types to be different from the default namespace for elements. Since relatively few users write schema-aware code, 99% of all type names in a typical stylesheet are in the XML schema namespace (for example `xs:integer`) and it makes sense to allow these to be written without a namespace prefix. For XSLT I propose to extend the `xpath-default-namespace` attribute so it can define both namespaces, space-separated. (Note however that when constructor functions are used, as in `xs:integer(@status)`, it is the default namespace for functions that applies.)

## 2.5. Named item types

In a stylesheet that uses maps to represent complex data structures, and especially when these are defined using the new `tuple()` syntax, you quickly find yourself using quite complex type definitions repeatedly on many different variable and function declarations. This has several disadvantages: it means that when the definition changes, code has to be changed in many different places; it fails to capture the semantic intent of the type; and it exposes details of the implementation that might be of no interest to the user.

I therefore propose to introduce the concept of named item types. These can be declared in a stylesheet using top-level declarations:

```
<xsl:item-type name="complex" as="tuple(r as xs:double, i as
xs:double)"/>
```

and can be referenced wherever an item type may appear using the syntax `type(type-name)`: for example `<xsl:param name="arg" as="type(complex)"/>`. Type names, like other names, are QNames, and if unprefixed are assumed to be in no namespace. The usual rules for import precedence apply. Types may be defined with visibility private or final; the definition cannot be overridden in another package.

Named item types also allow recursive type definitions to be created, for example:

```
<xsl:item-type name="binary-tree"
               as="tuple(left as type(binary-tree)?, value as item()*,
right as type(binary-tree)?)"/>
```

This means that item type names (like function names) are in scope within their own definitions. This creates the possibility of defining types that cannot be instantiated; I suggest that we leave implementors to issue warnings in such cases.

## 2.6. Type testing in patterns

With types becoming more expressive, and with increasing use of values other than nodes in `<xsl:apply-templates>`, the syntax `match=".[. instance of ItemType]"` to match items by their type becomes increasingly cumbersome. This syntax also has the disadvantage that there is no "smart" calculation of default priorities based on the type hierarchy. I therefore propose to introduce new syntax for patterns designed for matching items other than nodes.

- `type(T)` matches an item of type `T`, where `T` is a named item type. The default priority for such a pattern depends on the definition of `T`, and is the same as that of the pattern equivalent to `T`.
- A pattern in the form `atomic(EQName)`, followed optionally by predicates, matches atomic values of a specified atomic type. For example, `atomic(xs:string)[matches(., '[A-Z]*')]` matches all `xs:string` values comprising Latin upper-case letters.

  *Note, this syntax is needed because a bare EQName used as a pattern matches an element node with a given name. Semantically,* `atomic(Q)` *is equivalent to* `union(Q)` *(a singleton union).*
- Item types in the form `tuple(...)`, `map(...)`, `array(...)`, `function(...)`, or `union(...)` match any item that is an instance of the specified item type.

  In fact, for template rules that need to match JSON objects, a tuple type that names a selection of the fields in the object without giving their types will often be perfectly adequate: for example `match="tuple(ssn, first, middle, last, *)"` is probably enough to ensure that the right rule fires.

  The default priority for these patterns is defined later in the paper.

Any of these patterns may be followed by one or more predicates.

The effect of these changes is that for any `ItemType`, there is a corresponding pattern with the same or similar syntax:

- For the item type `item()`, the corresponding pattern is `.`
- For an item type expressed as an EQName `Q`, the corresponding pattern is `atomic(Q)`
- For an item type written as `type(...)`, `map(...)`, `array(...)`, `function(...)`, `tuple(...)`, or `union(...)`, the item type can be used as a pattern *as is*
- For an item type written as a `KindTest` (for example `element(P)` or `comment()`), the item type can be used as a pattern *as is* (this is because every `KindTest` is a `NodeTest`). There is one glitch here: as an item type, `node()`

matches all nodes, but as a pattern, it does not match attributes, namespace nodes, or document nodes. I therefore propose to introduce the syntax `node(*)`, which is defined to match any node (of any node kind) whether it is used as a step in a path expression or as the first step in a pattern.

These extensions to pattern syntax are designed primarily to make it easier to process the maps that result from parsing JSON using the recursive-descent template matching paradigm. For example, if the JSON input contains:

```
{ "ssn": "ABC12357", "firstName": "Michael", "dateOfBirth": "1951-10-11"}
```

then this can be matched by a template rule with the match pattern

```
match="tuple(ssn as xs:string, dateOfBirth, *)[?dateOfBirth castable as
xs:date]"
```

A possible extension, which I have not fully explored, is to allow nested patterns within a tuple pattern, rather than only allowing item types. For example, this would allow the previous example to be written:

```
match="tuple(ssn as xs:string, dateOfBirth[. castable as xs:date], *)"
```

Indeed, a further extension might be to allow a predicate wherever an item type is used, for example in the declaration of a variable or a function argument. While this is powerful, it creates considerable complications because of the fact that predicates can be context-sensitive

## 2.7. Function Conversion Rules

The so-called function conversion rules define how the supplied arguments to a function call are converted (where necessary) to the required type defined in the function signature. In XSLT (though not XQuery) the same rules are also used to convert the supplied value of a variable to its required type.

The name "function conversion rules" is rather confusing because the thing being converted is not necessarily a function, nor is the operation exclusively triggered by a function call, so my first proposal is to rename them "coercion rules". This is consistent with the way the term "function coercion" is already used in the spec.

The coercion rules are pragmatic and somewhat arbitrary: they are a compromise between the convenience to the programmer of not having to do manual conversion of values to the required type, and the danger of the system doing the wrong conversion if left to its own devices.

I propose to change the coercion rules so that where the required type is a derived atomic type (for example `xs:positiveInteger`), and the supplied value after atomization is an instance of the same primitive type (for example the `xs:integer` value 17) then the value is automatically converted -- giving a dynamic error, of course, if the conversion fails. Currently no-one uses the derived atomic types such as `xs:positiveInteger` in a function signature because of the inconvenience that you then can't supply the literal integer 17 in a function call. This change brings atomic values into line with the way that other values such as maps work: if a function declares the required type of a function argument as `map(xs:string, xs:integer)` then the caller can supply any map as an argument, and the function calling mechanism will simply check that the supplied map conforms with the constraints defined by the function for what kind of map it will accept; there is no need for the caller to do anything special to invoke a conversion.

*(I would have preferred a more radical change, whereby atomic values are labelled only with their primitive type, and not with a restricted type. So the expression 17 instance of `xs:positiveInteger` would return true, which is probably what most users would expect. However, I think this change would probably be too disruptive to existing applications.)*

I also propose to make a change to the way function coercion works. Function coercion applies when you supply a function `F` in a context where the required type is another function type `G`. The current rule is that this works provided that `F` accepts the arguments supplied in an actual call, and returns a value allowed by the signature of `G`; it doesn't matter whether `F` is capable of accepting everything that `G` accepts, so long as it accepts what is actually passed to it.

Currently function coercion fails if `F` and `G` have different arity. I propose to allow `F` to have lower arity than `G`; additional arguments supplied to `G` are simply dropped.

Consider how this might work for the higher-order function `fn:filter`, by analogy with the way it works in Javascript. Currently `fn:filter` expects as its second argument a function of type `$f as function(item()) as xs:boolean`. With this change to function coercion, we can extend this so the declared type is `$f as function(item(), xs:integer) as xs:boolean`. The extended version allows the predicate to accept a second argument, which is the position of the item in the sequence being filtered. But you can still supply a single-argument function; it just won't be told about the position.

The purpose of this change is to allow backwards-compatible extensions to higher-order functions; the information made available to the callback function can be increased without invalidating existing code.

## 2.8. Static type-checking rules

Some early XQuery developers favoured the use of "pessimistic static type checking", whereby a static type error is reported if any expression is not type-safe. (This is perhaps most commonly seen today in the implementation of XQuery offered with Microsoft's SQL Server database product.) More specifically, pessimistic static type checking signals an error unless the required type subsumes the supplied type. Experience has shown that pessimistic static type is rather inconvenient for most applications (especially as most applications are not schema-aware). XSLT fortunately steered clear of this area.[2]

The limited ability to perform "optimistic static type checking", whereby a static type error can be reported if the required type and the supplied type are disjoint, has been found to give considerable usability benefits; it is sufficient to detect a great many programming mistakes at compile time, provided that users are diligent in declaring the required types of variables and parameters, but it doesn't force the user to use verbose constructs (such as `treat as`) to enforce compile-time type safety.

I propose some modest changes to allow more obvious errors to be reported at compile time.

- First, I propose to allow a static type error to be reported in the case where the supplied type of an expression can satisfy the required type only in the event that its value is an empty sequence. For example, if the required type is `xs:integer*`, and the expression is a call on `xs:date()`, then it is not currently permitted to report a static error, because a call on `xs:date()` can yield an empty sequence, which would be a valid instance of the required type. In practice this situation is invariably a programmer mistake, and processors should be allowed to report it as such.

- Second, I propose introducing rules that allow certain path expressions (of the form `A/B`) to report an error if it is statically known that the result can only be an empty sequence. If the processor knows the node-kind of A, by means of static type inferencing, then it can report an error if B uses an axis that is always empty for that node kind: so `@A/@B` becomes a static error. (This error is suprisingly common, though it's not usually quite so blatant. It tends to happen when a template rule that only matches attributes does `<xsl:copy-of select="@*"/>`. Of course, this particular example is harmless, so we should reject it only if the stylesheet version is upped to 4.0).

  This ability is particularly useful in conjunction with schema-awareness. Users expect spelling mistakes in element names to be picked up by the compiler if the name used in the stylesheet is inconsistent with its spelling in the schema. Currently the language rules allow only a warning in this case.

[2]I have used the terms *optimistic* and *pessimistic* type checking for many years, but I cannot find any definitions in the literature. By *pessimistic static type checking* I mean what is often simply called *static* or *strict* type checking: a static error occurs if the inferred type of an expression is not a subtype of the type required for the context in which the expression is used. By contrast, I use *optimistic static type checking* to mean that a static error occurs only if the inferred type and the required type are disjoint (they have no values in common); in cases where the inferred type overlaps the required type, code is generated to perform run-time type checking.

- Third, an expression like `function($x){. + 3}` currently throws a dynamic error (`XPDY0002`) because the context item is absent. A strict reading of the XSLT specification suggests that the processor cannot report this as a compile time error (it only becomes an error if the function is actually evaluated). XQuery, it turns out, has fixed this (for named functions, though not for inline functions): it says that the static error XPST0008 can be raised in this situation. I propose changing XPDY0002 to be a type error, which means it can now be statically reported if detected during compilation, not just within function bodies, but in other contexts (such as `<xsl:on-completion>`) where there is no context item defined.

## 3. Functions

XSLT is a functional language, and version 3.0 greatly increases the role of functions by making them first-class objects and thus allowing higher-order functions. When you start to make extensive use of this capability, however, you start to encounter a few usability problems.

Firstly, the syntax for writing functions starts to become restrictive. You can either write global named functions in XSLT syntax, or local anonymous functions in XPath; neither syntax is particularly conducive to the very simple functions that you sometimes want to use in calls on `fn:filter()` or `fn:sort()`. It is also cumbersome to define a family of functions of different arity allowing some arguments to be omitted. I therefore propose to introduce some new syntax for writing functions.

### 3.1. Dot Functions

The syntax `.{EXPR}` is introduced as a shorthand for `function($x as item()) as item()* {$x ! EXPR}`. For example, this allows you to sort employees by last name then first name using the function call `sort(//employee, .{lastName, firstName})` where you would currently have to write `sort(//employee, function($emp) { $emp/lastName, $emp/firstName })`.

Exprience with other programming languages suggests that a more concise syntax for inline functions greatly encourages their use; indeed, we can imagine non-programmer users of XSLT mastering this syntax without actually understanding the concepts of higher-order functions.

### 3.2. Underscore Functions

In dot functions, we are limited to a single argument whose value is a single item (because that's the way the context item works). For the more general case, we introduce another notation: the underscore function. By way of an example, `_{$1 + $2}` is a function that takes two arguments (without declaring their type, so there are no constraints), and returns the sum of their values. This means that a function call such as `for-each-pair($seq1, $seq2, function($a1, $a2) {$a1 + $a2})` can now be written more concisely as `for-each-pair($seq1, $seq2, _{$1 + $2})`.

The arity of such a function is inferred from the highest-numbered parameter reference. Parameter references act like local variable references, but identify parameters by position rather than by name. There can be multiple references to the same parameter, and the function body doesn't need to refer to any parameters except the last (so the arity can be inferred). Parameters go "out of scope" in nested underscore functions.

The change to the function coercion rules means that if your function doesn't need to use the last argument, it doesn't matter that your function now has the wrong arity. For example, in a later section I propose an extension to the `<xsl:map>` instruction that provides an `on-duplicates` callback, which takes two values. To select the first duplicate, you can write <xsl:map on-duplicates="_{$1}"/>; to select the second, you can write <xsl:map on-duplicates="_{$2}"/>. Although the required type is a function with arity 2, you are allowed to supply a function that ignores the second argument.

Nested anonymous functions are perhaps best avoided in the interests of readability; but of course they are permitted. A numeric parameter reference such as $1 is not directly available in the closure of a nested function, but it can be bound to a conventional variable:

```
_{ let $x := $1, $g := _{$1 + $x} return $g(10) }(5)
```

## 3.3. Default Arguments

I propose to allow a single `<xsl:function>` declaration to define a family of functions, having the same name but different arity, by allowing parameters to have a default value. For example consider the declaration:

```
<xsl:function name="f:mangle" as="xs:integer">
  <xsl:param name="a" as="xs:string"/>
  <xsl:param name="options" as="map(*)" required="no" select="map{}"/>
  <xsl:sequence select="if ($options?upper) then upper-case($a) else
$a"/>
</xsl:function>
```

This declares two functions, `f:mangle#1` and `f:mangle#2`, with arity 1 and 2 respectively, based on whether the second argument is supplied or defaulted.

A parameter is declared optional with the attribute `required="no"`; if the attribute is optional, then its default value can be given with a `select` attribute. In the absence of a `select` attribute, the default value of an optional parameter is the empty sequence. A parameter can only be optional if all subsequent arguments are also optional.

The single `<xsl:function` declaration defines a set of functions having the same name, with arities in the range `M` to `N`, where `M` is the number of `<xsl:param>` elements with no default value, and `N` is the total number of `<xsl:param>` elements. The construct is treated as equivalent to a set of separate `xsl:function` declarations without optional parameters; for example, an overriding `xsl:function` declaration (one with higher import precedence, or one within an `xsl:override` element) might override one of these functions but not the others.

## 4. Conditionals

Conditional (if/then/else) processing can be done both in XPath and in XSLT. In both cases, for such a commonly used construct, the syntax is a little cumbersome. I believe that a few minor improvements can be made without difficulty and will be welcomed by the user community.

### 4.1. The `otherwise` operator

A common idiom in XPath is to see constructs like `(@discount, 0)[1]` to mean: take the value of the `@discount` attribute if present, or the default value 0 otherwise.

There are two drawbacks with this construct: firsly, unless you've come across it before, the meaning is far from obvious; and secondly, it only works if the first value is a singleton, rather than an arbitrary sequence.

I propose the syntax `@discount otherwise 0` as a more intuitive way of expressing this. The expression returns the value of the first operand, unless it is an empty sequence, in which case it returns the value of the second operand.

### 4.2. Adding `@select` to `<xsl:when>` and `<xsl:otherwise>`

Most XSLT instructions that allow a contained sequence constructor also allow a `select` attribute as an alternative. The `<xsl:when>` and `<xsl:otherwise>` elements are notable exceptions, and I propose to remedy this. For example this instruction:

```
<xsl:choose>
   <xsl:when test="@a=2">
      <xsl:sequence select="17"/>
   </xsl:when>
   <xsl:when test="@a=3">
      <xsl:sequence select="19"/>
   </xsl:when>
```

```
      <xsl:otherwise>
         <xsl:sequence select="23"/>
      </xsl:otherwise>
   </xsl:choose>
```

can be rewritten as:

```
   <xsl:choose>
      <xsl:when test="@a=2" select="17"/>
      <xsl:when test="@a=3" select="19"/>
      <xsl:otherwise select="23"/>
   </xsl:choose>
```

which makes it significantly more readable.

## 4.3. Adding `@then` and `@else` attributes to `<xsl:if>`

For the `xsl:if` instruction, rather than adding a `select` attribute, I propose to add two attributes, `then` and `else`. If either attribute is present then the contained sequence constructor must be empty. If one attribute is present and the other absent, the other defaults to () (the empty sequence).

This enables a construct like:

<xsl:if test="@a='yes' then="0" else="1"/>

This is likely to be particularly useful for delivering function results, in place of `xsl:sequence`; it will often enable a 2-way `xsl:choose` to be replaced with a 2-way `xsl:if`.

Consider this example from the XSLT 3.0 specification:

```
   <xsl:choose>
     <xsl:when test="system-property('xsl:version') = '1.0'">
       <xsl:value-of select="1 div 0"/>
     </xsl:when>
     <xsl:otherwise>
       <xsl:value-of select="xs:double('INF')"/>
     </xsl:otherwise>
   </xsl:choose>
```

which can (in all likelihood) be rewritten

```
   <xsl:if test="system-property('xsl:version') = '1.0'"
         then="1 div 0"
         else="xs:double('INF')"/>
```

Of course, we could also use an XPath conditional here. But when the expressions become a little longer, many users dislike using complex multi-line XPath expressions (partly because some editors ruin the layout, whereas editors offer good support for XML layout).

For another example, the function given earlier in this paper:

```
<xsl:function name="f:mangle" as="xs:integer">
  <xsl:param name="a" as="xs:string"/>
  <xsl:param name="options" as="map(*)" select="map{}"/>
  <xsl:sequence select="if ($options?upper) then upper-case($a) else
$a"/>
</xsl:function>
```

can now be written:

```
<xsl:function name="f:mangle" as="xs:integer">
  <xsl:param name="a" as="xs:string"/>
  <xsl:param name="options" as="map(*)" select="map{}"/>
  <xsl:if test="$options?upper" then="upper-case($a)" else="$a"/>
</xsl:function>
```

## 4.4. `xsl:message/@test` attribute

Users have become familiar with the ability to "compile out" instructions using a static `use-when` expression, for example

```
<xsl:message use-when="$debug"/>
```

Currently this only works if `$debug` is a static variable; if it becomes necessary to use a non-static variable instead, the construct has to change to the much more cumbersome

```
<xsl:if test="$debug">
   <xsl:message/>
</xsl:if>
```

I propose that `<xsl:message>` should have a `test` attribute, bringing it into line with `<xsl:assert>`.

Verbose wrapping of instructions in `<xsl:if>` is also seen when constructing output elements, for example one might see a long sequence of instructions of the form:

```
<xsl:if test="in:maturity-date">
   <out:maturityDate>{maturity-date}</out:maturityDate>
</xsl:if>
```

I considered proposing that all instructions should have a `test` or `when` attribute, defining a condition which allows the instruction to be skipped. Having experimented with such a capability, however, I'm not convinced it improves the language.

## 4.5. Equality Operators

There are in effect four different equality operators for comparing atomic values, all with slightly different rules:

- The "=" operator is implicitly existential, and converts untyped atomic values to the type of the other operand: this leads to curiosities such as the fact that `A = B` being different from `not(A = B)`, and to non-transitivity (if `X` is `xs:untypedAtomic`, then `X = '4'` and `X = 4` can both be true, but `4 = '4'` gives a type error).
- The "eq" operator eliminates the existential behaviour, and converts untyped atomic values to strings. This avoids some of the worst peculiarities of the "=" operator, but the type promotion rules mean that it edge cases, it is still not transitive. The result of the operator is context-sensitive; for example the result of comparing two `xs:dateTime` values can depend on the implicit timezone.

  The comparison performed by `xsl:sort` and `xsl:merge` is based on the "eq" and "le" operators, but NaN is considered equal to itself. The lack of transitivity with edge cases involving mixed numeric types creates a potential security weakness in that it might be possible to construct an artificial input sequence to `xsl:sort` that causes the instruction not to terminate.
- The operator used by the `deep-equal()` function, and also (by reference) by `distinct-values()`, `index-of()`, `fn:sort()`, and `<xsl:for-each-group>`, differs from "eq" primarily in that it returns false rather than throwing an error when comparing unrelated types; it also compares `NaN` as equal to itself. Because it handles conversion among numeric types in the same way as "eq", it is still non-transitive in edge cases, which is particularly troublesome when the operator is used for sorting or grouping. Like "eq", the result is context-sensitive.
- The "same key" operator used implicitly for comparing keys in maps (for example in `map:contains()`) is designed to be error-free, context-free, and transitive. So it always returns false rather than throwing an error; the result is never context-sensitive; and it is always transitive.

It's difficult to sort all of this out while retaining an adequate level of backwards compatibility, but I propose that:

- Type promotion when comparing numeric types should be changed to use the rules of the "same key" operator throughout. In effect this means that all numeric comparisons are done by converting both operands to infinite-precision `xs:decimal` (with special rules for infinity and NaN). This change makes "eq" transitive. Although this creates a minor backwards incompatibility in edge cases, I believe this change can be justified on security grounds; the current rules mean there is a risk that sorting will not terminate for some input sequences. These rules extend to other functions that compare numeric values, for example `min()` and `max()`, but the promotion rules for arithmetic are unchanged: adding an `xs:double` and an `xs:decimal` still delivers an xs:double.

- All four operators should handle timezones in the way that the "same key" operator does: that is, a date/time value with a timezone is not considered comparable to one without. This change makes the result of a comparison independent of the dynamic context in which it is invoked, which enables optimizations that are disallowed in 3.0 simply because of the remote possibility that the input data will contain a mix of timezoned and untimezoned dates/times.

  This change is perhaps more significant from the perspective of backwards compatibility, and perhaps there needs to be a 3.0-compatible mode of execution that retains the current behaviour.

## 5. Template Rules and Modes

Template rules and modes are at the heart of the XSLT processing model. The `xsl:mode` declaration in XSLT 3.0 usefully provides a central place to define options and properties for template rule processing. Packages also help to create better modularity. But anyone who has to debug a large complex stylesheet with 20 or more modules knows what a nightmare it can be to find out where a particular bit of logic is located, so further improvements are possible.

### 5.1. Enclosed Modes

I propose to allow template rules to be defined by using `xsl:template` as a child of `xsl:mode`. An xsl:mode declaration that contains template rules is referred to as an enclosed mode. Such template rules must have no `mode` attribute (it defaults to the name of the containing mode). They must also have no `name` attribute. If a mode is an enclosed mode, then all template rules for the mode must appear within the `xsl:mode` declaration, other than template rules declared using `xsl:override` in a different package. Specifying `mode="#all"` on a template rule outside the enclosed mode is interpreted as meaning "all modes other than enclosed modes". The default mode for `xsl:apply-templates` instructions within the enclosed mode is the enclosing mode itself.

This feature is designed to make stylesheets more readable: it becomes easier to get an overview of what a mode does, and it becomes easier to find the template rules associated with a mode. It makes it easier to copy-and-paste a mode from one stylesheet to another. It means that to find the rules for a mode, there are fewer places you need to look: the rule will either be within the mode itself, or (if the mode is not declared `final`) within an `xsl:override` element in a using package.

To further encourage the use of very simple template rules, I propose allowing `xsl:template` to have a `select` attribute in place of a sequence constructor. This allows for example:

```
<xsl:mode name="border-width" as="xs:integer">
<xsl:template match="aside" select="1"/>
<xsl:template match="footnote" select="2"/>
<xsl:template match="*" select="0"/>
</xsl:mode>
```

A template rule with a `select` attribute must not contain any `xsl:param` or `xsl:context-item` declarations.

## 5.2. Typed modes

It is often the case that all template rules in a mode return the same type of value, for example nodes, strings, booleans, or maps. This is almost a necessity, since anyone writing an `xsl:apply-templates` instruction needs to have some idea of what will be returned.

I propose therefore that the `xsl:mode` declaration should acquire an `as` attribute, whose value is a sequence type. If present, this acts as the default for the `as` attribute in `xsl:template` rules using that mode. Individual template rules may have an `as` attribute that declares a more precise type, but only if it is a true subtype.

The presence of this attribute enables processors to infer a static type for the result of the `xsl:apply-templates` instruction.

In the interests of forcing good practice, the `xsl:mode/ @as` attribute is required in the case of an enclosed mode.

## 5.3. Default Namespace for Elements

Anyone who follows internet programming forums such as StackOverflow will know that the number one beginner mistake with XSLT is to assume that an unprefixed name, used in a path expression or match pattern, will match an unprefixed element name in the source document. In the presence of a default namespace declaration, of course, this is not the case.

What's particularly annoying about this problem is that the consequences bear no obvious relationship to the nature of the mistake. It generally means that template rules don't fire, and path expressions don't select anything. Those are tough symptoms for beginners to debug, when they have no idea where to start looking.

It's worth noting that only a minority of documents actually use multiple namespaces, and in those that do, there is rarely any ambiguity in the sets of local names used. It's therefore unsurprising that beginners imagine that namespaces are something they can learn about later if they need to.

The `xpath-default-namespace` attribute in XSLT 2.0 was an attempt to tackle this problem; but unfortunately it only solved the problem if you already knew that the problem existed.

I want to propose a more radical solution:

- Unprefixed element names in path expressions and match patterns should match by local name alone, regardless of namespace; that is, `NNNN` is interpreted as `*:NNNN`.

  This is a radical departure and for backwards compatibility, it must be possible to retain the status quo. My guess is that the vast majority of stylesheets will still work perfectly well with this change.
- The syntax `:local` (with a leading colon) becomes available to force a no-namespace match, regardless of default namespace.
- The option to match by local name can be explicitly enabled (for any region of the stylesheet) by specifying `xpath-default-namespace="##any"`, while the option for unprefixed names to match no-namespace names can be selected by setting the attribute to either a zero-length string (as in XSLT 3.0) or, for emphasis, to `"##local"` (a notation borrowed from XSD).
- The "default default" for `xpath-default-namespace` becomes implementation-defined, with a requirement that it be configurable; implementors can choose how to configure it, and what the default should be. (This includes the option to use the default namespace declared in the source document, if known).

This gives implementors the option to provide beginners with an interface in which unprefixed element names match the way that beginners expect: by local name only. Users who understand namespaces can then switch to the current behaviour if they wish, or can qualify all names (using the new syntax `:name` for no-namespace names if necessary), to make sure that the problem does not arise.

This proposal is also motivated by the challenges posed by the way namespaces are handled in HTML5. The HTML5 specification defines a variation on the XPath 1.0 specification that changes the way element names in path expressions match. The proposal to make unprefixed element names match (by default) by local name alone removes the need for HTML5 to get special treatment.

## 6. Processing Maps and Arrays

The introduction of maps and arrays into the data model has enabled more complex applications to be written in XSLT, as well as allowing JSON to be processed alongside XML. But experience with these new features has revealed some of their limitations, and a second round of features is opportune.

## 6.1. Array construction

The XSLT instruction `xsl:array` is added to construct an array.

The tricky part is how to construct the array members (in general, a sequence of sequences). The same problem exists for the square and curly array constructors in XPath, and I propose to solve the problem in the same way.

First I propose a new function

```
array:of((function() as item()*)*) => array(*)
```

which takes a sequence of zero-arity functions as its input, and evaluates each of those functions to return one member of the array. For example

```
array:of((_{1 to 5}, _{7 to 10}))
```

returns the array `[(1,2,3,4,5), (7,8,9,10)]`

(The underscore syntax for writing simple functions – in this case, zero-arity functions – was described earlier in the paper).

For a more complex example,

```
array:of(for $x in 1 to 5 return _{1 to $x})
```

returns the array `[(1), (1,2), (1,2,3), (1,2,3,4), (1,2,3,4,5)]`.

Now I propose an instruction `xsl:array` that accepts either a `select` attribute or a contained sequence constructor, and processes the resulting sequence in the same way as the `array:of()` function, with one addition: any item in the result that is not a zero-arity function is first wrapped in a zero-arity function. For example:

```
<xsl:array select="1 to 5"/>
```

returns the array `[1,2,3,4,5]`; while

```
<xsl:array>
  <a/>
  <b/>
  <c/>
</xsl:array>
```

returns the array `[<a/>, <b/>, <c/>]`, and

```
<xsl:array select="1, 2, 3, _{}, ${4,5,6}"/>
```

returns the array `[1, 2, 3, (), (4,5,6)]`

## 6.2. Map construction

The `<xsl:map>` instruction acquires an attribute `on-duplicates`. The value of the attribute is an XPath expression that evaluates to a function; the function is called when duplicate map entries are encountered. For example, `on-duplicates="_{$1}"` selects the first duplicate, `on-duplicates="_{$2}"` selects the last, `on-duplicates="_{$1, $2}"` combines the duplicates into a single sequence, and `on-duplicates="_{string-join(($1, $2), '|')}"` concatenates the values as strings with a separator.

## 6.3. The Lookup Operator ("?")

In 3.0, the right-hand side of the lookup operator (in both its unary and binary versions) is restricted to be an NCName, an integer, the token "*", or a parenthesized expression.

To provide slightly better orthogonality, I propose relaxing this by allowing (a) a string literal, and (b) a variable reference. In both cases the semantics are equivalent to enclosing the value in parentheses: for example `$array?$i` is equivalent to `$array?($i)` (which can also be written `$array($i)`), and `$map?"New York"` is equivalent to `$map?("New York")` (which can also be written `$map("New York")`).

## 6.4. Iterating over array members

The lookup operator `$array?*` allows an array to be converted to a sequence, and often this is an adequate way of iterating over the members of the array. But where the members of the array are themselves sequences, this loses information: the result of `array{(1,2,3), (4,5,6)}?*` is `(1,2,3,4,5,6)`.

To make processing such arrays easier, I introduce a new clause for FLWOR expressions: `for member $var in array-expression` which binds `$var` to each member of the array returned by the *array-expression*, in turn.

For example:

```
for member $var in array{(1,2,3), (4,5,6)} return sum($var)
```

returns `(6, 15)`

As with `for` and `let`, I allow `for member` as a free-standing expression in XPath.

Currently the only way to achieve such processing is with higher-order functions: `array:for-each($array, sum#1)`.

We can also consider an XSLT instruction `<xsl:for-each-member>` but the question becomes, how should the current member be referenced? I'm no great enthusiast for yet more `current-XXX()` functions, but stylistic consistency is important, and this certainly points to the syntax:

```
<xsl:for-each-member select="array{(1,2,3), (4,5,6)}">
  <total>{sum(current-member())}</total>
</xsl:for-each-member>
```

## 6.5. Rule-based recursive descent with maps and arrays

The traditional XSLT processing model for transforming node trees relies heavily on the interaction of the `xsl:apply-templates` instruction and match patterns.

The model doesn't work at all well for maps and arrays, for a number of reasons. The reasons include:

- We don't have convenient syntax for matching maps and arrays in patterns; all we have is general predicates, which are cumbersome to use.
- Because there is no parent or ancestor axis available when processing maps and arrays, a template rule for processing part of a complex structure cannot get access to information from higher in the structure unless it is passed down in the form of parameters. In addition, there is no mechanism for defining a template rule to match a map or array in a way that is sensitive to the context in which it appears.
- There is no built-in template corresponding to the shallow-copy template that works effectively for maps and arrays, allowing the stylesheet author to define rules only for the parts of the structure that need changing
- Template rules always match items. But with a map, the obvious first level of decomposition is not into items, but into entries (key-value pairs). Similarly, with arrays, the first level of decomposition is into array members, which are in general sequences rather than single items.

The following sections address these issues in turn.

### 6.5.1. Type-based pattern matching

In 3.0 it is possible to use a pattern of the form `match=".[. instance of T]"` to match items by their type. This syntax is clumsy, to say the least. I therefore propose some new kinds of patterns with syntax closely aligned with item type syntax. The following new kinds of pattern are introduced (by example):

- `atomic(xs:date)`
  Matches an atomic value of type `xs:date`.
- `union(xs:date, xs:dateTime, xs:time)`
  Matches an atomic value belonging to a union type.
- `map(xs:string, element())`
  Matches a map belonging to a map type.
- `tuple(first, middle, last)`
  Matches a map belonging to a tuple type.
- `array(xs:integer)`
  Matches an array whose members are of a given type
- `type(T)`
  Matches an an item belonging to a named type (declared using `xsl:item-type`).

In each case the item type can be followed by predicates. For example, strings starting with "#" can be matched using the pattern `atomic(xs:string)[starts-with(., '#')]`, while tuples representing female employees might be matched with the pattern `tuple(ssn, lastName, firstName, *)[?gender='F']`

The following rules are proposed for the default priority of these patterns (in the absence of predicates):

- For patterns corresponding to the generic type `function(*)` the priority is -0.75; for `map(*)` and `array(*)` it is -0.5.
- For atomic patterns such as `atomic(xs:string)`, the priority is 1 - $0.5^N$, where N is the depth of the type in the type hierarchy. For example, `xs:decimal` is 0.5, `xs:integer` is 0.75, `xs:long` is 0.875. In all cases the resulting priority is between zero and one.

  `atomic(xs:anyAtomicType)` gets a priority of 0.

  The rule extends to user-defined atomic types.

  The rule ensures that if S is a subtype of T, then the priority of S is greater than the priority of T.
- For union patterns such as `union(xs:integer, xs:date)`, the priority is the product of the priorities of the atomic member types. So for this example, the priority is 0.375.

  Again, this rule ensures that priorities reflect subtype relationships: for example `union(xs:integer, xs:date)` has a lower priority than `atomic(xs:integer)` but a higher priority than `union(xs:decimal, xs:date)`.

  The rule does not ensure, however, that overlapping types have equal priority; for example when matching an integer, the pattern `union(xs:integer, xs:date, xs:time)` will be chosen in preference to `union(xs:integer, xs:double)`. The rules will not, therefore, be a reliable way of resolving ambiguous matches.
- For a specific array type `array(M)`, the priority is the normalized priority of the item type of *M* (the cardinality of *M* is ignored). Normalized priority is calculated as follows: if the priority is *P*, then the normalized priority is *(P +1)/2*. That is, base priorities in the range -1 to +1 are compressed into the range 0 to +1.
- For a specific map type `map(K, V)`, the priority is the product of the normalized priorities of *K* and the item type of *V* (the cardinality of *V* is ignored).
- For a specific function type `function(A1, A2, ...) as V`, the priority is the product of the normalized priorities of the item types of the arguments. The cardinalities of the argument types, and the result type, are ignored.

Enterprising users may choose to exploit the fact that `function(xs:integer)` has a higher priority than `function(xs:decimal)` as a way of implementing polymorphic function despatch.

- For a non-extensible tuple type `tuple(A as t1, B as t2, ...)`, the priority the product of the normalized priorities of the item types of the defined fields.
- For an extensible tuple type `tuple(A as t1, B as t2, ..., *)`, the priority is -0.5 plus (0.5 times the priority of the corresponding non-extensible tuple type).

  This rule has the effect that an extensible tuple type is never considered for a match until all non-extensible tuple types have been eliminated from consideration.

Like the existing rules for the default priority of node patterns, these rules are a little rough-and-ready, and will not always give the result that is intuitively correct. However, they follow the general principle that selective patterns have a higher priority than non-selective patterns, so it's likely that they will resolve most cases in the way that causes least surprise. When things get complex, users can always define explicit priorities.

The existing rules for node patterns often ensure that overlapping rules have the same priority, thus leading to warnings or errors when more than one pattern matches. That remains true for the new rules when predicates are used, but in the absence of predicates, there are many cases where overlapping patterns do not have the same priority.

The most important use case for the new kinds of pattern is to match maps (objects) when processing JSON input, and in this case using tuples that name the distinguishing fields/properties of each object should achieve the required effect, regardless whether extensible or inextensible tuple types are used.

### 6.5.2. Decomposing Maps

I propose a function `map:entries($map)` which returns a sequence of maps, one per key-value pair in the original map. The map representing each entry contains the following fields:

- key: the key (an atomic value)
- value: the associated value (any sequence)
- container: the map from which this entry was extracted.

That is, the result matches the type `tuple(key as xs:anyAtomicType, value as item()*, container as map(*))`. To process a map using recursive-descent template rule processing, it is possible to use an instruction of the form `<xsl:apply-templates select="map:entries($map)"/>`, and then to process each entry in the map using a separate template rule. The presence of the `container` field compensates for the absence of an ancestor axis: it gives access to entries in the containing map other than the one being processed. For example:

```
<xsl:template match="tuple(key, value)[?key='ssn']">
   <xsl:if test="?container?location='London'" then="'UK'||?value"
else="'US'||?value"/>
</xsl:template>
```

This makes the immediate context of a map entry available to the called template rule. For more distant context, it is generally necessary to pass the information explicitly, typically using tunnel parameters. (Navigating further back using multiple container steps is feasible in theory, but clumsy in practice.)

An alternative to use of tunnel parameters is to add information to the map being processed: instead of `<xsl:apply-templates select="map:entries($map)"/ >`, you can write `<xsl:apply-templates select="$map:entries($map) ! map:put(., 'country-name': $country)"/>`, and the extra data will then be available in the called templates as `?country-name`.

## 7. New Functions

In this section, I propose various new or enhanced functions to add to the core function library, based on practical experience. (Other new functions, such as `array-of()`, have been proposed earlier in the paper).

### 7.1. `fn:item-at`

The function `fn:item-at($s, $i)` returns the same result as `fn:subsequence($s, $i, 1)`. It is useful in cases where the positional filter expression `$s[EXPR]` is unsuitable because the subscript expression `EXPR` is focus-dependent.

### 7.2. `fn:stack-trace`

I propose a new function `fn:stack-trace()` to return a string containing diagnostic information about the current execution state. The detailed content and format of the returned string is implementation-dependent.

I also propose a standard variable `$err:stack-trace` available within `xsl:catch` to contain similar information about the execution state at the point where a dynamic error occurred.

## 7.3. `fn:deep-equal` with options

An extra argument is added to `fn:deep-equal`; it is a map following the "option parameter conventions". The options control how the comparison of the two operands is performed. Options should include:

- Ignore whitespace text nodes
- Normalize whitespace in text and attribute nodes
- Treat comments as significant
- Treat processing instructions as significant
- Treat in-scope namespace bindings as significant
- Treat namespace prefixes as significant
- Treat type annotations as signficant
- Treat `is-ID`, `is-IDREF` and `nillable` properties as signficant
- Treat all nodes as untyped
- Use the "same key" comparison algorithm for atomic values (as used for maps), rather than the "eq" algorithm
- Ignore order of sibling elements

## 7.4. `fn:differences()`

A new function, like `fn:deep-equal()`, except that rather than returning a true or false result, it returns a list of differences between the two input sequences. If the result is an empty sequence, the inputs are deep-equal; if not, the result contains a sequence of maps giving information about the differences. The map contains references to nodes within the tree that are found to be different, and a code indicating the nature of the difference, plus a narrative explanation. The specification will leave the exact details implementation-defined, but standardised in enough detail to allow applications to generate diagnostics.

For example, `fn:differences(<a x='3'/ >, <a x='4'/ >)` might return `map{0: $1/ @x, 1: $2/ @x, 'code': 'different-string-value', 'explanation': "The string value of the @x attribute differs ('3' vs '4')"}`

The values of entries 0 and 1 here are references to the attribute nodes in the supplied input sequences.

## 7.5. `fn:index-where($input, $predicate)`

Returns a sequence of integers (monotonically ascending) giving the positions in the the input sequence where the predicate function returns true.

Example: `subsequence($in, 1, index-where($in, .{exists(self::h1)}))` returns the subsequence of the input up to and including the first `h1` element.

Equivalent to

`(1 to count($input)) [$predicate(subsequence($input, ., 1)]`

### 7.6. `fn:items-before()`, `fn:items-until()`, `fn:items-from()`, `fn:items-after()`

These new higher-order functions all take two arguments: an input sequence, and a predicate that can be applied to items in the sequence to return a boolean.

If `N` is the index of the first item in the input sequence that matches the predicate, then:

- `fn:items-before()` returns items with `position() lt N`
- `fn:items-until()` returns items with `position() le N`
- `fn:items-from()` returns items with `position() ge N`
- `fn:items-after()` returns items with `position() gt N`

### 7.7. `map:index($input, $key)`

Returns a map in which the items in `$input` are indexed according to the atomized value of the `$key` function. For example `map:index(// employee, .{@location})` returns a map `$M` such that `$M?London` will return all employees having `@location='London'`.

The `$key` function may return a sequence of values in which case the corresponding item from the input will appear in multiple entries in the index.

### 7.8. `map:replace($map, $key, $action)`

If the map `$map` contains an entry for `$key`, the function calls `$action` supplying the existing value associated with that key, and returns a new map in which the value for the key is replaced with the result of the `$action` function.

If the map contains no entry for the `$key`, calls `$action` supplying an empty sequence, and returns a new map containing all existing entries plus a new entry for that key, associated with the value returned by the `$action` function.

For example, `map:replace($map, 'counter', _{($1 otherwise 0) + 1})` sets the value of the `counter` entry in the map to the previous value plus 1, or to 1 if there is no existing value (and returns the new map).

### 7.9. `fn:highest()` and `fn:lowest()`

Currently given as example user-written functions in the 3.1 specification, these could usefully become part of the core library. For example, `highest(// p, string-length#1)` returns the longest paragraph in the document.

### 7.10. `fn:replace-with()`

The new function `fn:replace-with($in, $regex, $callback, [$flags])` is similar to `fn:replace()`, but it computes the replacement string using a callback function. For example, `replace-with($in, '[0-9]+', .{string(number() +1)})` adds one to any number appearing within the supplied string: "Chapter 12" becomes "Chapter 13".

### 7.11. `fn:characters()`

Splits a string into a sequence of single-character strings. Avoids the clumsiness of `string-to-codepoints(x)!codepoints-to-string()`.

### 7.12. `fn:is-NaN()`

Returns true if and only if the argument is the `xs:float` or `xs:double` value `NaN`.

### 7.13. Node construction functions

Once you start using higher-order functions extensively, you discover the problem that in order for a user-written function to create nodes, your code has to be written in XSLT rather than in XPath. This is restrictive, because it means for example that the logic cannot be included in static expressions, nor in expressions evaluated using `xsl:evaluate` (I've seen people using `fn:parse-xml()` to get around this restriction, for example `fn:parse-xml("<foo/>")` to create an element node named `foo`). A set of simple functions for constructing new nodes would be very convenient. Specifically:

- `fn:new-element(QName, content)` – constructs a new element node with a given name; `$content` is a sequence of nodes used to form the content of the element, following the rules for constructing complex content.
- `fn:new-attribute(QName, string)` – constructs a new attribute node, similarly
- `fn:new-text(string)` - constructs a new text node
- `fn:new-comment(string)` - constructs a new comment node
- `fn:new-processing-instruction(string, string)` - constructs a new processing instruction node

- `fn:new-document(content)` - constructs a new document node node
- `fn:new-namespace(content)` - constructs a new namespace node

Despite their names, these functions are defined to be *non-deterministic with respect to node identity*: if called twice with the same arguments, it is system-dependent whether or not you get the same node each time, or two different nodes. In practice, very few applications are likely to care about the difference, and leaving the system to decide leaves the door open for optimizations such as loop-lifting.

Here's an example to merge the attributes on two sequences of elements, taken pairwise:

```
<out>
  <xsl:sequence select="for-each-pair($seq1, $seq2,
                                          _{new-element(node-name($1),
($1/@*, $2/@*))})"/>
</out>
```

The functional approach to node construction is useful when elements are created conditionally. Consider this example from the XSLT 3.0 specification:

```
<xsl:for-each-group select="node()"
            group-adjacent="self::ul or self::ol">
    <xsl:choose>
        <xsl:when test="current-grouping-key()">
            <xsl:copy-of select="current-group()"/>
        </xsl:when>
        <xsl:otherwise>
            <p>
                <xsl:copy-of select="current-group()"/>
            </p>
        </xsl:otherwise>
    </xsl:choose>
</xsl:for-each-group>
```

This can now be written:

```
<xsl:for-each-group select="node()"
            group-adjacent="self::ul or self::ol">
  <xsl:if test="current-grouping-key()"
          then="current-group()"
          else="new-element(QName("", "p"), current-group())"/>
</xsl:for-each-group>
```

## References

[1] Michael Kay. *Transforming JSON using XSLT 3.0*. Presented at XML Prague, 2016. Available at `http://archive.xmlprague.cz/2016/files/`

xmlprague-2016-proceedings.pdf and at http://www.saxonica.com/papers/xmlprague-2016mhk.pdf

[2] Michael Kay. *An XSD 1.1 Schema Validator Written in XSLT 3.0*. Presented at Markup UK, 2018. Available at http://markupuk.org/2018/Markup-UK-2018-proceedings.pdf and at http://www.saxonica.com/papers/markupuk-2018mhk.pdf

[3] Michael Kay, John Lumley. *An XSLT compiler written in XSLT: can it perform?*. Presented at XML Prague, 2019. Available at http://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf and at http://www.saxonica.com/papers/xmlprague-2019mhk.pdf

[4] John Lumley, Debbie Lockett and Michael Kay. *Compiling XSLT3, in the browser, in itself*. Presented at Balisage: The Markup Conference 2017, Washington, DC, August 1-4, 2017. In Proceedings of Balisage: The Markup Conference 2017. Balisage Series on Markup Technologies, vol. 19 (2017). Available at https://doi.org/10.4242/BalisageVol19.Lumley01

[5] *XSL Transformations (XSLT) Version 3.0*. W3C Recommendation, 8 June 2017. Ed. Michael Kay, Saxonica. http://www.w3.org/TR/xslt-30