XMLmind XML Editor - Support of XPath 1.0

Hussein Shafie
XMLmind Software
<xmleditor-support@xmlmind.com>

XMLmind XML Editor - Support of XPath 1.0

Hussein Shafie
XMLmind Software
<xmleditor-support@xmlmind.com>

Publication date November 15, 2024

Abstract

XPath 1.0 is used everywhere in XMLmind XML Editor (**XXE** for short): to configure the editor, to script commands and even in CSS stylesheets. This document contains the reference for all the XPath 1.0 extension functions supported by **XXE**.

This document also contains the reference for *XED*, a very small, very simple scripting language based on XPath 1.0. Because XED allows to modify in place the document being edited, it can be used to script advanced macro-commands.

Table of Contents

List of Examples

Part I. Native XPath 1.0 support

XMLmind XML Editor (**XXE** for short) natively supports XPath 1.0. This XPath 1.0 implementation, based on the XPath engine of XT, James Clark's XSLT processor, is small, fast, fully conformant and features many extension functions.

Table of Contents

Chapter 1. XPath functions

All the standard XPath 1.0 functions are supported: `boolean`, `ceiling`, `concat`, `contains`, `count`, `false`, `floor`, `id`, `lang`, `last`, `local-name`, `name`, `namespace-uri`, `normalize-space`, `not`, `number`, `position`, `round`, `starts-with`, `string`, `string-length`, `substring`, `substring-after`, `substring-before`, `sum`, `translate`, `true`.

The following XSLT 1.0 functions are also supported: `current`, `document`, `format-number`, `system-property`, `key`, `generate-id`, `function-available`, `element-available`, `unparsed-entity-uri` with the following specificities:

- In `document(`*`relative_URI`*`)`, *`relative_URI`* is not resolved against the URI of the XSLT stylesheet (because there is no such XSLT stylesheet).
- The 3-argument form of `format-number()` is not supported.
- `key()` always returns an empty node-set when used outside a XED script [9] or a Schematron.
- `element-available()` returns `true` for any element name in the "`http://www.w3.org/1999/XSL/Transform`" namespace and `false` otherwise.
- `unparsed-entity-uri()` always returns an empty string.
- `system-property()` supports the following XSLT 1.0 properties: `xsl:version`, `xsl:vendor`, `xsl:vendor-url`, and also the following XSLT 2.0 properties: `xsl:product-name`, `xsl:product-version`, in addition to Java™'s system properties.

1. Extension functions

node-set copy(*`node-set`*)

Returns a deep copy of specified node set.

object defined(string *`variable-name`*, *`default-value`*?)

When passed a single argument, returns `true()` if a variable having specified name is defined; returns `false()` otherwise.

When passed two arguments, returns the value of the variable having specified name if this variable is defined; returns *`default-value`* otherwise.

`variable-name` must have one of the following forms: *`prefix`*:*`local_part`*, where *`prefix`* has been defined in the document being edited, or {*`namespace_URI`*}*`local_part`*.

node-set difference(*`node-set1`*, *`node-set2`*)

Returns a node-set containing all nodes found in *`node-set1`* but not in *`node-set2`*.

boolean ends-with(*`string1`*, *`string2`*)

Returns true if string *`string1`* ends with string *`string2`*. Returns false otherwise.

number index-of-node(*`node-set1`*, *`node-set2`*)

Returns the rank of a node in *`node-set1`*. The node which is searched in *`node-set1`* is specified using *`node-set2`*: it is first node in *`node-set2`* (which generally contains a single node). The index of first node in *`node-set1`* is 1 and not 0. Returns -1 if the searched node is not found in *`node-set1`*.

object if(boolean *test1*, object *value1*, ..., boolean *testN*, object *valueN*, ..., object *fallback*)

Evaluates each *testi* in turn as a boolean. If the result of evaluating *testi* is true, returns corresponding *valuei*. Otherwise, if all *testi* evaluate to false, returns *fallback*.

Example:

```
if(@x=1,"One",@x=2,"Two",@x=3,"Three","Other than one two three")
```

node-set intersection(*node-set1*, *node-set2*)

Returns a node-set containing all nodes found in both *node-set1* and *node-set2*.

boolean is-editable(*node-set*?)

Returns `true()` if first node of specified node set is editable; returns `false()` otherwise. When *node-set* is not specified, this function is applied to the context node.

Also returns `true()` if specified node set is empty.

`is-editable()` is a convenient alternative to:

```
not(ancestor-or-self::*[¬
  property('{http://www.xmlmind.com/xmleditor/namespace/property}readOnly')])
```

See `property()` [5].

string join(node-set *node-set*, string *separator*)

Converts each node in *node-set* to a string and joins all these strings using *separator*. Returns the resulting string.

Example: `join(//h1, ', ')` returns `"Introduction, Conclusion"` if the document contains 2 `h1` elements, one containing `"Introduction"` and the other `"Conclusion"`.

string lower-case(*string*)

Returns the value of its argument after translating every character to its lower-case correspondent as defined in the appropriate case mappings section in the Unicode standard.

boolean matches(string *input*, string *pattern*, string *flags*?)

Similar to XPath 2.0 function `matches`. Returns true if *input* matches the regular expression *pattern*; otherwise, it returns false.

Note that unless ^ and $ are used, the string is considered to match the pattern if any substring matches the pattern.

Optional *flags* may be used to parametrize the behavior of the regular expression:

`m`

Operate in multi-line mode. In multi-line mode, the expressions ^ and $ match just after or just before, respectively, a line terminator or the end of the input sequence. By default, these expressions only match at the beginning and the end of the entire input sequence.

`s`

Operate in single line mode. In single line mode, the expression . matches any character, including a line terminator. By default, this expression does not match line terminators.

`i`

Operate in case-insensitive mode (in a manner consistent with the Unicode Standard).

`l`

Treat the pattern as a sequence of literal characters.

Regular expression reference: `java.util.regex.Pattern`.

Examples: `matches("foobar", "^f.+r$")` returns true. `matches("CamelCase", "ca", "i")` returns true.

number max(*`node-set`*), number max(*`number`*, ..., *`number`*)

The first form returns the maximum value of all nodes of specified node set, after converting each node to a number.

Nodes which cannot be converted to a number are ignored. If all nodes cannot be converted to a number, returns `NaN`.

The second form returns the maximum value of all specified numbers (at least 2 numbers).

Arguments which cannot be converted to a number are ignored. If all arguments cannot be converted to a number, returns `NaN`.

number min(*`node-set`*), number min(*`number`*, ..., *`number`*)

Same as max() but returns the minimum value of specified arguments.

number pow(*`number1`*, *`number2`*)

Returns *`number1`* raised to the power of *`number2`*.

string property(string *`property-name`*, *`node-set`*?)

Returns the application-level property having specified name attached to first node of specified node set. When *`node-set`* is not specified, this function is applied to the context node.

Returns the empty string if the specified node set is empty or if the first node in the node set does not have specified property.

`property-name` must have one of the following forms: *`prefix`*:*`local_part`*, where *`prefix`* has been defined in the document being edited, or {*`namespace_URI`*}*`local_part`*. Examples:

- `foo`,
- `bar:foo`, where prefix `bar` is bound to "`http://www.bar.com/ns`" in the document being edited,
- `{}foo`,
- `{http://www.xmlmind.com/xmleditor/namespace/property}sourceURL`,
- `{http://www.xmlmind.com/xmleditor/namespace/property}readOnly`,
- `{http://www.xmlmind.com/xmleditor/namespace/property}configurationName`.

See also `is-editable()` [4].

string replace(string *`input`*, string *`pattern`*, string *`replacement`*, string *`flags`*?)

Similar to XPath 2.0 function `replace`. Returns the string that is obtained by replacing all non-overlapping substrings of *`input`* that match the given *`pattern`* with an occurrence of the *`replacement`* string.

The *`replacement`* string may use $1 to $9 to refer to captured groups.

Optional *`flags`* may be used to parametrize the behavior of the regular expression:

`m`

Operate in multi-line mode. In multi-line mode, the expressions `^` and `$` match just after or just before, respectively, a line terminator or the end of the input sequence. By default, these expressions only match at the beginning and the end of the entire input sequence.

`s`

Operate in single line mode. In single line mode, the expression `.` matches any character, including a line terminator. By default, this expression does not match line terminators.

`i`

Operate in case-insensitive mode (in a manner consistent with the Unicode Standard).

`l`

Treat the pattern as a sequence of literal characters.

Example: `replace("foobargeebar", "b(.+)r", "B$1R")` returns `"fooBaRgeeBaR"`.

string resolve-uri(string *`uri`*, string *`base`*?)

If *`uri`* is an absolute URL, returns *`uri`*.

If *`base`* is specified, it must be a valid absolute URL, otherwise an error is reported.

If *`uri`* is a relative URL,

- if *`base`* is specified, returns *`uri`* resolved using *`base`*;
- if *`base`* is not specified, returns *`uri`* resolved using the base URL of the context node.

If *`uri`* is the empty string,

- if *`base`* is specified, returns *`base`*;
- if *`base`* is not specified, returns the base URL of the context node.

string relativize-uri(string *`uri`*, string *`base`*?)

Converts absolute URL *uri* to an URL which is relative to specified base URL *base*. If *base* is not specified, the base URL of the context node is used instead.

Uri must be a valid absolute URL, otherwise an error is reported. If *`base`* is specified, it must be a valid absolute URL, otherwise an error is reported.

Example: returns `"../john/.profile"` for *uri*=`"file:///home/john/.profile"` and *base*=`"file:///home/bob/.cshrc"`.

If *uri* cannot be made relative to *base* (example: *uri*=`"file:///home/john/public_html/index.html"` and *base*=`"http://www.xmlmind.com/index.html"`), *uri* is returned as is.

string serialize(*`node-set`*)

Serializes specified node-set and returns a well-formed, parseable, XML string. This string is not nicely indented. This string always starts with `<?xml version="1.0"?>` as it is intended to be directly consumed by other commands such as `paste`.

If multiple nodes are to be serialized (as opposed to a single element node or to a document node), these nodes are first wrapped in a `{http://www.xmlmind.com/xmleditor/namespace/clipboard}clipboard` element.

Note that some node-sets cannot be serialized: the empty node-set, node-sets containing just attribute nodes, node-sets mixing a document node with other kind of nodes, etc. In such cases, an error is reported.

node-set tokenize(string *`input`*, string *`pattern`*, string *`flags`*?)

Similar to XPath 2.0 function `tokenize`, except that it returns a node-set comprising text nodes rather than a sequence of strings.

This function breaks the *`input`* string into a sequence of strings, treating any substring that matches *`pattern`* as a separator. The separators themselves are not returned. If *`input`* is the zero-length string, the result is an empty node-set.

Optional *`flags`* may be used to parametrize the behavior of the regular expression:

m

Operate in multiline mode.

i

Operate in case-insensitive mode.

Examples: `tokenize("abracadabra", "(ab)|(a)")` returns a node-set containing 6 text nodes. The string values of these text nodes are `""`, `"r"`, `"c"`, `"d"`, `"r"`, `""`. `tokenize("ABRACADABRA", "(ab)|(a)", "i")` returns a node-set containing 6 text nodes. The string values of these text nodes are `""`, `"R"`, `"C"`, `"D"`, `"R"`, `""`.

string upper-case(*`string`*)

Returns the value of its argument after translating every character to its upper-case correspondent as defined in the appropriate case mappings section in the Unicode standard.

string uri-or-file-name(*`string`*)

Converts specified string to an URL. Specified string may be an (absolute) URL supported by XMLmind XML Editor or the absolute or relative filename of a file or of a directory. An error is reported if the argument cannot be converted to an URL.

string uri-to-file-name(*`string`*)

Converts specified argument, a "`file://`" URL, to a native file name. Returns the empty string if argument is not a "`file://`" URL.

2. Java™ methods as extension functions

A call to a function `ns:foo` where `ns` is bound to a namespace of the form `java:`*`className`* is treated as a call of the static method `foo` of the class with fully-qualified name *`className`*. Example:

```
xmlns:file="java:java.io.File"

file:createTempFile('xxe', '.tmp')
```

Hyphens in method names are removed with the character following the hyphen being upper-cased. Example:

```
file:create-temp-file('xxe', '.tmp')
```

is equivalent to:

```
file:createTempFile('xxe', '.tmp')
```

A non-static method is treated like a static method with the this object as an additional first argument. Example:

```
file:delete-on-exit(file:createTempFile('xxe', '.tmp'))
```

A constructor is treated like a static method named `new`. Example:

```
xmlns:url="java:java.net.URL"

url:new('http://www.xmlmind.com/xmleditor/')
```

Overloading based on number of parameters is supported; overloading based on parameter types is not. Example, it is possible to invoke:

```
url:new('http://www.xmlmind.com/xmleditor/')
```

though both `java.net.URL(java.lang.String spec)` and `java.net.URL(java.net.URL context, java.lang.String spec)` exist. It is not possible to invoke:

```
file:new('.')
```

because both `java.io.File(java.lang.String pathname)` and `java.io.File(java.net.URI uri)` exist.

Extension functions can return objects of arbitrary types which can then be passed as arguments to other extension functions.

Types are mapped between XPath and Java™ as follows:

XPath type	Java™ type
string	`java.lang.String`
number	double
boolean	boolean
node-set	`com.xmlmind.xml.xpath.NodeIterator`

On return from an extension function, an object of type `com.xmlmind.xml.doc.XNode` is also allowed and will be treated as a node-set; also any numeric type is allowed and will be converted to a number.

Part II. The XED scripting language

XED is a very small, very simple scripting language, leveraging the native XPath 1.0 implementation of XMLmind XML Editor, allowing to modify in place an XML document.

The initial context node of a XED script

A XED script modifies in place a single XML document. A XED script always has an XPath context node belonging to the document being modified. The initial context node is set by the environment running the XED script. This initial context node is always the document itself ("`/`") when the script is run by **Paste As** → **Paste from Word Processor or Browser** or by "`xmltool indent -script` in *The xmltool command-line utility*".

Table of Contents

Chapter 2. Language syntax

1. Syntax

XED scripts are found in text files (recommended filename extension "`.xed`") having the following syntax:

```
script -> [ encoding ]
          [ namespace | include | command | if | foreach | macro ]*

encoding -> 'encoding' charset_string ';'

namespace -> 'namespace' [ prefix_NCName '=' ] namespace_URI_string ';'

include -> 'include' script_URI_string ';'

command -> command_name_NCName '(' [ argument ]* ')' ';'

argument -> XPath_expression | XML_template

if -> 'if' test '{' block_contents '}'
      [ 'elseif' test '{' block_contents '}' ]*
      [ 'else' '{' block_contents '}' ]

test -> boolean_XPath_expression

block_contents -> [ command | if | foreach ]*

foreach -> 'for-each' node_set_XPath_expression '{' block_contents '}'

macro -> macro_name_NCName '(' [ parameters ] ')' '{' block_contents '}'

parameters -> parameter_name_NCName [ ',' parameter_name_NCName ]*
```

- A *`string`* is a quoted XPath string which may contain XML character entities. Example: `'Hello &apos;world&#39;!'`.
- An *`XPath_expression`* is an XPath 1.0 expression, where the concept of default namespace is supported for element names and where strings may contain XML character entities.
- An *`XML_template`* is a literal XML element, possibly containing XPath 1.0 expressions delimited by curly braces ("`{ }`"). See Section 10, “XML templates” [15].

2. Text file encoding

```
encoding -> 'encoding' charset_string ';'
```

The encoding of the text file containing a XED script may be specified using an `encoding` declaration found at the very beginning of the file. Examples (note that the charset names are case-insensitive):

```
encoding 'UTF-8';

encoding 'iso-8859-1';
```

If such `encoding` declaration is missing, the encoding is automatically determined using the byte order marks (**BOM**) found at the beginning of the text file. If the encoding cannot be determined this way, then it is assumed to be the encoding used by the operating system (e.g. "`Windows-1252`" on a computer running Windows with a western locale).

3. Comments

Multi-line comments are strings delimited by "`(:`" and "`:)`". Comments may be nested. Example:

```
(: Hello
   World! :)
```

Note that it's not possible to use "`(: :)`" comments inside XPath expressions and inside XML templates.

4. Including a script file with `include`

```
include -> 'include' script_URI_string ';'
```

Including script file `B.xed` into script file `A.xed` is strictly equivalent to replacing in file `A.xed` the `include` directive by the contents of file `B.xed`.

Because macro commands [14] are local to a script file, `include` is mainly useful to include the same set of macro commands into several script files.

Note that it's also possible to run a script file, possibly with a different context node, using command `script` [26].

5. Namespace declarations

```
namespace -> 'namespace' [ prefix_NCName '=' ] namespace_URI_string ';'
```

Declares a namespace and its associated prefix.

If the prefix is absent, then the declared namespace is the default namespace of elements. This default namespace applies to element names found in XPath expressions and in XML templates [15], but not to attribute and variable names.

Examples:

```
namespace "http://www.w3.org/1999/xhtml";

namespace html = "http://www.w3.org/1999/xhtml";

namespace g="urn:x-mlmind:namespace:group";
```

6. Commands

```
command -> command_name_NCName '(' [ argument ]* ')' ';'

argument -> XPath_expression | XML_template
```

Invokes command called *command_name_NCName* with specified arguments.

Predefined commands are documented in Chapter 3, *Predefined commands* [17]. A user may define her own commands using `macro` [14].

Examples:

```
set-variable("next",
             following-sibling::node()[1][self::text() and
                                          starts-with(., "&#xA;")]);
delete-text("^\n", "", $next);

replace(<g:envelope>{translate(., " ", " ")}</g:envelope>);

unwrap-element();
```

7. Conditional processing with `if`

```
if -> 'if' test '{' block_contents '}'
      [ 'elseif' test '{' block_contents '}' ]*
      [ 'else' '{' block_contents '}' ]

test -> boolean_XPath_expression

block_contents -> [ command | if | foreach ]*
```

The *`test`* tests found after `if`, `elseif`, ..., `elseif` are evaluated in turn until a test evaluates as `true()`. When this happens, the evaluation of tests stops there and the *`block_contents`* following the successful test is run. If no test evaluates as `true()`, then the *`block_contents`* following `else`, if any, is run.

Examples:

```
if $table-container and count($table-container//td) = 1 {
    replace(copy(.), $table-container);
}

if ./node() {
    unwrap-element();
} else {
    delete();
}

if @href {
    remove-attribute("name");
} elseif ./node() {
```

```
    unwrap-element();
} else {
    delete();
}
```

8. Repetition with `for-each`

```
foreach -> 'for-each' node_set_XPath_expression '{' block_contents '}'

block_contents -> [ command | if | foreach ]*
```

XPath expression *`node_set_XPath_expression`* is evaluated as a node set, then `for-each` iterates over the nodes of this set, running *`block_contents`* at each iteration.

Inside *`block_contents`*, the context node is the current node of the node set iterated over. Example:

```
(: The context node is /html/body. :)

for-each .//p {
    (: Inside this for-each, the context-node is a p
       which is a descendant of body. :)

    remove-attribute("style");
}
```

9. Macro commands

```
macro -> macro_name_NCName '(' [ parameters ] ')' '{' block_contents '}'

parameters -> parameter_name_NCName [ ',' parameter_name_NCName ]*

block_contents -> [ command | if | foreach ]*
```

A macro command is simply a user-defined command. It is invoked by its name just like predefined commands [17]. It can be passed arguments just like predefined commands. Beside its parameters which act like local variables, it can have other local variables declared by the means of special command `variable` [30]. Like any other command, a `macro` is executed in the context of the current context node[1].

A macro command is local to the script file containing it. See `include` [12] to learn how the same macro command may be shared between several script files.

Examples:

```
macro unstyle-heading(heading) {
    variable("text", string(.));

    for-each $heading//*[(self::b or
                          self::big or
```

[1]The context node of the script or the context node of the body of a `for-each`.

```
                          self::cite or
                          self::dfn or
                          self::em or
                          self::i or
                          self::q or
                          self::s or
                          self::small or
                          self::strong or
                          self::tt or
                          self::u) and
                          string(.) = $text and
                          not(@id)] {
        unwrap-element();
    }
}

macro heading-to-bridgehead() {
    set-attribute("class",
                  concat("bridgehead", substring-after(local-name(), "h")));
    set-element-name("p");
}
```

10. XML templates

A command [13] may be passed XPath expressions or XML templates as its arguments. An *XML template* is a literal XML element, possibly containing XPath 1.0 expressions delimited by curly braces ("`{ }`"). Examples:

```
<db:formalpara><db:title>TITLE HERE</db:title></db:formalpara>

<a href="{concat('#', $id)}" title="{$title}" class="xref"/>

<g:envelope>{normalize-space(.)}</g:envelope>
```

The enclosed XPath expressions are evaluated as *strings* in the context of the current context node[1]. This means that these enclosed expressions must be found inside attribute values, text, comment or processing-instruction nodes.

If you want attribute values, text, comment or processing-instruction nodes to actually contain curly braces, then you must escape these curly braces by doubling them (that is, "`{`" becomes "`{{`" and "`}`" becomes "`}}`").

Note that whitespace is significant inside an XML template. Therefore do not indent XML templates.

An XML template is copied and its enclosed XPath expressions, if any, are substituted with their values, each time the template is passed as an argument to a command. Therefore, an XML template creates a new element each time it is used.

Sometimes, you want to pass a command a list of nodes rather than a single element. When this is the case, use a `g:envelope` element, where "`g`" is the prefix of namespace "`urn:x-ml-mind:namespace:group`", as a container for these nodes. Example: replace context node (`.`) by a text node containing the value of its `title` attribute:

```
namespace g = "urn:x-mlmind:namespace:group";

...

replace(<g:envelope>{@title}</g:envelope>);
```

Chapter 3. Predefined commands

1. break

```
break()
```

Exit from the enclosing `for-each` loop.

Example:

```
for-each $elementList/* {
    ...
    if @xml:id = "i2" {
        break();
    }
}
```

2. continue

```
continue()
```

Skip what follows in the enclosing `for-each` loop and continue with next iteration.

Example:

```
for-each $elementList/* {
    if @xml:id = "i1" {
        continue();
    }
    ...
}
```

3. delete

```
delete(xnodes?)
```

Delete specified nodes (whatever their types) or attributes. Parameter *`xnodes`* defaults to the context node.

Examples:

```
delete();

delete(/html/head/comment());

delete(//span[@s:class = "dummy"]);

delete(//@s:*);
```

4. delete-text

```
delete-text(pattern, flags?, from?)
```

Delete text matching regular expression *`pattern`* from node *`from`*. Parameter *`from`* defaults to the context node. The text is deleted no matter the node containing it: node *`from`*, descendants of node *`from`* or a mix between node *`from`* and its descendants.

Optional *`flags`* may be used to parametrize the behavior of the regular expression:

`a`
: Delete *all* occurrences of *`pattern`*.

`m`
: Operate in multi-line mode. In multi-line mode, the expressions `^` and `$` match just after or just before, respectively, a line terminator or the end of the input sequence. By default, these expressions only match at the beginning and the end of the entire input sequence.

`s`
: Operate in single line mode. In single line mode, the expression `.` matches any character, including a line terminator. By default, this expression does not match line terminators.

`i`
: Operate in case-insensitive mode (in a manner consistent with the Unicode Standard).

`l`
: Treat the pattern as a sequence of literal characters.

Regular expression reference: `java.util.regex.Pattern`.

Examples:

```
delete-text("Note:\s*");

delete-text("\n", "as");

delete-text("^\s+", "", $div);
```

5. delete-key

```
delete-key(key_name, key_value?)
```

Delete entry *`key_value`* from map *`key_name`*. If *`key_value`* is not specified, delete map *`key_name`*.

Parameter *`key_name`* is a string representing an XML qualified name. Parameter *`key_value`* is a string.

See also XSLT function `key()` and command `update-key()` [29].

Examples:

```
delete-key("refs", "introduction");

delete-key("refs");
```

6. error

```
error(message_part, ..., message_part)
```

Concatenate all the arguments after converting them to strings, then print the resulting error message on the console. The execution of the script is stopped after this.

See also commands `warning()` [30] and `message()` [25].

Examples:

```
error("FAILED!");

error("Expected ", $expectedCount, ", got ", $count);
```

7. group

```
group()
```

Command `group()` groups under a common parent element all the sibling elements having the same *group mark* (which is attribute `g:id`).

Text, comment and processing-instruction nodes found between elements having the same group mark are automatically added to the common parent element.

Using command `group()` is a two step process:

1. Add the following attributes to the elements you want to group: `g:id`, `g:container`, `g:nesting`, where "`g`" is the prefix of namespace "`urn:x-mlmind:namespace:group`".

2. Invoke command `group()`.

Command `group()` traverses the document modified by the script. It processes separately “sections” containing marked elements. What we call a “section” here is simply any element directly containing at least one child element having a `g:container` attribute.

When done, command `group()` automatically removes all `g:id`, `g:container`, `g:nesting` attributes from the document.

Attribute	Type	Role
`g:id`	Any non empty string.	Elements having the same `g:id` belong to the same group, hence will be given a common container parent. This attribute is required for `group()` to work.

`g:container`	String having the following format:
	container -> *element_qname*
	[*attribute*]*
	attribute -> *attr_qname* '='
		Specifies the container parent. Examples:
		`ul`
		`ol type='A' compact='compact'`
		`div class="role-section{$nesting}"`

Attribute	Type	Role
	attr_value *attr_value* -> '"' *string* '"' | "'" *string* "'" *string* may contain variable "{$nesting}" which is substituted with the actual nesting level of the group to be created. This actual nesting level is an integer starting at 1.	This attribute is required for `group()` to work.In principle, suffice to set `g:container` on the first member of a group. However it's often simpler, and it does not harm, to set `g:container` on all the members of a group.

`g:nesting`	Positive number. Default value: 0.	This number allows to specify how groups nest in each other. Example:
		
		`<li g:id="a" g:container="ul" g:nesting="1">`
		`  ...`
		`</li>`
		`<li g:id="b" g:container="ul" g:nesting="3.14">`
		`  ...`
		`</li>`
		
		First `li` will be made a child of an `ul` created by `group()`. Second `li`, because 3.14 is greater than 1, will be made a child of a second `ul`. This second `ul` will be nested into the first `ul`. Note that the exact values of `g:nesting` do not matter as these numbers are simply compared to each others.
		
		When set on an element having no `g:id`, this attribute allows to end groups.
		
		Because the default value of `g:nesting` is 0, any element having no `g:nesting` and no `g:id` attributes automatically ends all groups.

Example 3.1. Basic use of command `group()`

In `samples/group1.xhtml`, convert sibling `p` elements starting with "`1)`", "`2)`", "`3)`", etc, to proper lists.

```
<p>1) Item #1.</p>
<p>2) Item #2.</p>
<p>3) Item #3.</p>
```

You can run the following script, `samples/group1.xed`, using `samples/make_samples.bat` (Windows) or `samples/make_samples` (Linux, Mac) . The resulting file is `samples/out1.xhtml`.

```
namespace "http://www.w3.org/1999/xhtml";
namespace g="urn:x-mlmind:namespace:group";
```

```
for-each //p[matches(., "^\d+\)\s*")] {
    delete-text("^\d+\)\s*");

    set-element-name("li");

    set-attribute("g:id", "numbered");
    set-attribute("g:container", "ol style='list-style-type: upper-roman;");
}

group();
```

Example 3.2. Creating nested groups

In `samples/group2.xhtml`, convert sibling `p` elements starting with "`*`", "`**`", "`***`", etc, to proper lists. These lists may be nested. For example, a `p` element starting with "`**`" is to be contained in a list nested in the list containing `p` elements starting with "`*`".

```
<p>* First item.</p>
<p>* Second item.</p>

<p>** Nested first item.</p>

<p>*** Nested, nested first item.</p>
<p>*** Nested, nested second item.</p>

<p>** Nested second item.</p>
...
```

You can run the following script, `samples/group2.xed`, using `samples/make_samples.bat` (Windows) or `samples/make_samples` (Linux, Mac) . The resulting file is `samples/out2.xhtml`.

```
namespace "http://www.w3.org/1999/xhtml";
namespace g="urn:x-mlmind:namespace:group";

for-each //p[matches(., "^\*+\s+")] {
    set-variable("label", substring-before(., " "));
    set-variable("bullets", count(tokenize($label, "\*")) - 1);
    message("label=&quot;", $label, "&quot;, bullets=", $bullets);

    delete-text("^\*+\s+");

    set-element-name("li");

    set-attribute("g:id", concat("bulleted", $bullets));
    set-attribute("g:container", "ul");
    set-attribute("g:nesting", $bullets);
}

group();
```

Example 3.3. Group members having different element types

Variant of Example 3.1, “Basic use of command `group()`” [20]. In `samples/group3.xhtml`, we want “continuation paragraphs” to be part of the current list item rather than end the current list group:

```
<p>1) Item #1.</p>
<p>Continuation paragraph.</p>

<p>2) Item #2.</p>
<p>Continuation paragraph.</p>

<p>3) Item #3.</p>

<p>Not a continuation paragraph.</p>
```

You can run the following script, `samples/group3.xed`, using `samples/make_samples.bat` (Windows) or `samples/make_samples` (Linux, Mac) . The resulting file is `samples/out3.xhtml`.

```
namespace "http://www.w3.org/1999/xhtml";
namespace g="urn:x-mlmind:namespace:group";

for-each //p[matches(., "^\d+\)\s*")] {
    delete-text("^\d+\)\s*");

    set-variable("listItem", <li/>);
    wrap-element($listItem);❶

    set-attribute("g:id", "numbered", $listItem);
    set-attribute("g:container", "ol", $listItem);
}

for-each //p[preceding-sibling::li[position()=last() and @g:id] and
             following-sibling::li[position()=1 and @g:id]] {❷
    set-variable("listItem", preceding-sibling::li[last()]);

    set-attribute("g:id", $listItem/@g:id);
}

group();
```

❶ We could have used:

```
set-element-name("li");
```

just like in `samples/group1.xed`. The above variant wraps the `p` into an `li` rather than changing the `p` to an `li`.

❷ Detect “continuation paragraphs” and give them the same `g:id` as the current list item. That makes these paragraphs members of the current list group.

Normally `out3.xhtml` should contain:

```
<ol>
  <li><p>Item #1.</p></li>
  <p>Continuation paragraph.</p>
  <li><p>Item #2.</p></li>
  <p>Continuation paragraph.</p>
  <li><p>Item #3.</p></li>
</ol>
```

which is invalid. However `out3.xhtml` actually contains:

```
<ol>
  <li><p>Item #1.</p><p>Continuation paragraph.</p></li>
  <li><p>Item #2.</p><p>Continuation paragraph.</p></li>
  <li><p>Item #3.</p></li>
</ol>
```

This works because command `group()` knows that a `p` element cannot be a child of an `ol` element[1]. When this occurs, `group()` will attempt to add the “alien group member” at the end of the preceding group member rather than adding it at the end of the group container.

8. insert-after

```
insert-after(inserted, after?)
```

Insert *`inserted`*, one or more nodes, after node *`after`*. Parameter *`after`* defaults to the context node.

Examples:

```
insert-after(copy(.));

insert-after(<meta name="{$metaName}" content="{$metaContent}"/>, $title);
```

Always pass *detached nodes* as the first argument of `insert` or `replace`

The nodes passed as the first argument of commands `insert-after`, `insert-before`, `insert-into` or `replace` must have *no parent nodes* (“detached nodes”).

Example: something like what follows works fine:

```
delete($caption);
insert-after($caption, $table);
```

While something like what follows will report an execution error:

```
insert-after($caption, $table);
delete($caption);
```

When passing detached nodes is not possible or not desirable, simply use XPath extension function `copy()` [3] to copy the nodes. Example:

[1] Command `group()` uses the schema of the document edited by the script to determine that.

```
insert-after(copy($caption), $table);
delete($caption);
```

9. insert-before

```
insert-before(inserted, before?)
```

Insert *inserted*, one or more nodes, before node *before*. Parameter *before* defaults to the context node.

Important note: Always pass *detached nodes* as the first argument of `insert` or `replace` [23].

Examples:

```
insert-before(copy(.));

insert-before(<meta charset="UTF-8"/>, $title);
```

10. insert-into

```
insert-into(inserted, first?, into?)
```

Insert *inserted*, one or more nodes, into node *into*, as its first children if *first* is `true()` and as its last children if *first* is `false()`. Parameter *first* defaults to `false()`. Parameter *into* defaults to the context node.

Important note: Always pass *detached nodes* as the first argument of `insert` or `replace` [23].

Examples:

```
insert-into(copy($captionPara/node());

insert-into(copy($captionPara/node(), true());

insert-into(<b>Note: </b>, true(), $p);
```

11. invoke

```
invoke(class_name, any_argument, ..., any_argument)
```

Invokes a command written in Java™ other than the predefined ones. This command creates an instance of specified class name before invoking this instance. Class *class_name* must extend `com.xmlmind.xml.xed.Command`.

Example:

```
invoke("com.xmlmind.xmleditext.paste_from_word.engine.BeforeSave");
```

The class name may be followed by a number of arguments in order to invoke a specific constructor rather that the default one. The syntax for this is:

```
specific_constructor_invocation -> class_name '(' S [ arg_list ]? S ')'
arg_list -> arg [ S ',' S arg S ]*
arg -> 'true' | 'false' | integer_number | double_number |
       'null' | double_quote_string | single_quote_string
```

Literal `null` denotes the null string. `"\""` escaped double quotes are supported in *`double_quote_string`*. `'\''` escaped single quotes are supported in *`single_quote_string`*.

Example:

```
invoke("com.xmlmind.xmleditext.paste_from_word.engine.BeforeSave(false, false)");
```

12. message

```
message(message_part, ..., message_part)
```

Concatenate all the arguments after converting them to strings, then print the resulting information message on the console.

See also commands `error()` [19] and `warning()` [30].

Examples:

```
message("SUCCESS!");

message("Found ", serialize($found));
```

13. replace

```
replace(replacement, replaced?)
```

Replace node *`replaced`* by *`replacement`*, one or more nodes. Parameter *`replaced`* defaults to the context node.

Important note: Always pass *detached nodes* as the first argument of `insert` or `replace` [23].

Examples:

```
replace(copy(./node()));

replace(<img src="{translate(@src, '\', '/')}"
             alt="{if(@o:title != '', @o:title, '???')}"/>,
        $parent);
```

14. remove-attribute

```
remove-attribute(name, element?)
```

Remove attribute having specified name from specified element. Parameter *name* is a string representing an XML qualified name. Parameter *element* defaults to the context node.

Example:

```
remove-attribute("xml:id");

remove-attribute("lang", parent::*);
```

15. remove-property

```
remove-property(name, node?)
```

Remove property having specified name from specified node. Parameter *name* is a string representing an XML qualified name. Parameter *node* defaults to the context node.

A *property* is an application-level attribute which can be set on all kind of nodes (that is, not only on elements) and which is not serialized.

Examples:

```
remove-property("hidden");

namespace my="urn:x-acme:namespace:local";

remove-property("my:status", /);
```

16. save-document

```
save-document(file_name, indented?)
```

Save the document being modified by the XED script to specified file. A relative filename is relative to the current working directory. Parameter *indented*, which defaults to `true()`, specifies whether the save file should be indented.

This command is mainly useful to debug complex XED scripts.

Examples:

```
save-document("c:\temp\debug.xml");

save-document("out.xml", (:indent:) false());
```

17. script

```
script(script_location, context_node?)
```

Run the XED script found at specified location. Parameter *script_location* must be an absolute or relative URI. A relative URI is relative to the location of the XED script containing the `script()` command. Parameter *context_node* defaults to the current context node.

XED scripts run this way are loaded and cached once for all. Therefore there is almost no performance penalty in using command `script()`, and this, even inside `for-each` loops.

Examples:

```
script("prune.xed");

script("utils/arrange.xed", .//db:index);
```

18. set-attribute

```
set-attribute(name, value, element?)
```

In specified element, set attribute having specified name to specified value. Parameter *`name`* is a string representing an XML qualified name. Parameter *`element`* defaults to the context node.

Examples:

```
set-attribute("class", "note");

set-attribute("xml:id", generate-id());

set-attribute("g:id", "numbered", $listItem);
```

19. set-doctype

```
set-doctype(doc_name, public_id, system_id, internal_subset?)
```

Add, change or remove `<!DOCTYPE>` in the document being modified by the XED script.

When parameter *`internal_subset`* is not specified or is the empty string, the `<!DOCTYPE>` will have no internal subset.

When parameters *`doc_name`*, *`public_id`* and *`system_id`* are all passed the empty string, the `<!DOCTYPE>` is removed from the document.

Examples:

```
set-doctype("html", "-//W3C//DTD XHTML 1.0 Strict//EN", "xhtml1-strict.dtd");

set-doctype("html", "-//W3C//DTD XHTML 1.0 Strict//EN", "xhtml1-strict.dtd",
            '<!ENTITY xxe "XMLmind XML Editor">');

set-doctype("", "", "");
```

20. set-element-name

```
set-element-name(name, element?)
```

Change the name of element *element* to name *name*. Parameter *name* is a string representing an XML qualified name. The default namespace [12], if any, is taken into account when parsing this qualified name. Parameter *element* defaults to the context node.

Examples:

```
set-element-name("li");

set-element-name("html:caption", preceding-sibling::title);
```

Parameter *name* may be followed optionally by attributes. When this is the case, the name of element *element* is changed to specified value and specified attributes are added or replaced to/in element *element*.

Example:

```
set-element-name("li style='color: #333;' class='item'");
```

21. set-property

```
set-property(name, value, node?)
```

In specified node, set property having specified name to specified value. Parameter *name* is a string representing an XML qualified name. Parameter *node* defaults to the context node.

A *property* is an application-level attribute which can be set on all kind of nodes (that is, not only on elements) and which is not serialized.

Examples:

```
set-property("hidden", "yes");

namespace my="urn:x-acme:namespace:local";

set-property("my:status", "DRAFT", /);
```

22. set-variable

```
set-variable(name, value)
```

Set variable having specified name to specified value. Parameter *name* is a string representing an XML qualified name.

The variable set by this command is the variable *in scope* having specified name. It is either a variable local to a `macro` [14] or a global variable. See also `command variable()` [30].

Examples:

```
namespace my="urn:x-acme:namespace:local";

set-variable("my:list", <ul/>);
```

```
set-variable("listItem", preceding-sibling::li[last()]);
```

23. translate-chars

```
translate-chars(from, to, node?)
```

Remap characters in the text contained in specified node and all its descendants. Parameter *node* defaults to the context node.

A character found in string *from* is replaced by the character at the corresponding position in string *to*. If there is a character in string *from* with no character at a corresponding position in string *to* (because string *from* is longer than string *to*), then the occurrences of that character are removed.

Examples:

```
translate-chars("/", "\");

translate-chars(" -.", "_");

translate-chars(" ", " ", $title);
```

24. update-key

```
update-key(key_name, key_value, keyed_node?)
```

Add node or attribute *keyed_node* to the entry having value *key_value* of map *key_name*. Map *key_name* is created if needed to. Entry *key_value* is created if needed to.

Parameter *key_name* is a string representing an XML qualified name. Parameter *key_value* is a string. Parameter *keyed_node* defaults to the context node.

See also XSLT function `key()` and command `delete-key()` [18].

Example:

```
update-key("anchors", @name);

update-key("refs", substring-after(@href, "#"), .);

update-key("ids", @name, $ancestorP);
```

25. unwrap-element

```
unwrap-element(element?)
```

Replace specified element by its child nodes. Parameter *element* defaults to the context node.

Examples:

```
unwrap-element();

unwrap-element(//span[not(@style)]);
```

26. variable

```
variable(name, value)
```

Declares a local variable having specified name in a `macro` [14]. Initializes local variable to specified value. Parameter *name* is a string representing an XML qualified name.

It's an error to use `variable()` outside a `macro`.

Example:

```
set-variable("v1", 0);

macro m1() {
    (: Local to macro m1. Shadows global variable v1. :)
    variable("v1", 1000);

    set-variable("v1", $v1 + 500);

    if ($v1 != 1500) {
        error("FAILED");
    }
}

if ($v1 != 0) {
    error("FAILED");
}
```

27. warning

```
error(message_part, ..., message_part)
```

Concatenate all the arguments after converting them to strings, then print the resulting warning message on the console.

See also commands `error()` [19] and `message()` [25].

Examples:

```
warning("FIXME");

warning("Expected ", $expectedCount, ", got ", $count);
```

28. wrap-element

```
wrap-element(container, transfer_attributes?, element?)
```

Move element *element* at the end of element *container* then replace *element* by *container*. Parameter *element* defaults to the context node.

If parameter *transfer_attributes*, which defaults to `false()`, is passed `true()`, then the attributes of element *element* are transferred to element *container*. This attribute transfer removes each attribute of element *element* in turn and if the removed attribute is not already set in element *container*, it adds this attribute to *container*.

Example:

```
wrap-element($listItem);

wrap-element($listItem, false(), .);

wrap-element(<blockquote/>, (:transfer attributes:) true());
```

Index