XSH2 Reference

Table of Contents

1. XSH2 Language

XSH2 acts as a command interpreter. Individual commands must be separated with a semicolon. In the interactive shell, backslash may be used at the end of a line to indicate that a command continues on the next line. Output redirection can be used to pipe output of some XSH command to some external program, or to capture the output to a variable. See Redirection for more info.

XSH2 command help provides a complete reference, instantly from the command-line:

`help command` gives a list of all XSH2 commands.

`help type` gives a list of all argument types.

`help topic` followed by documentation chapter gives more information on a given topic.

`help toc` displays the table of contents.

1.1. Files/Documents

XSH2 is designed as an environment for querying and manipulating XML and HTML documents. Use open or create commands to load an XML or HTML document from a local file, external URL (such as http:// or ftp://), string or pipe. XSH2 can optionally validate the document during parse process (see validation and load-ext-dtd). Parsed documents are stored in memory as DOM [http://www.w3.org/DOM/] trees, that can be navigated and manipulated with XSH2 commands and XPath language, whose names and syntax make working with the DOM tree a flavor of working in a UNIX filesystem.

A parsed document is usually stored in a variable. XSH2 shares variables with the XPath engine, so if e.g. `$doc` is a XSH2 variable holding a document (or, more generally any node-set), then `$doc//section/title` is an XPath expression selecting all `title` subelements of all `section` elements within the (sub)tree of $doc.

Although XSH2 is able to parse remote documents via `http://` or `ftp://`, it is only able to save them locally. To upload a document to a remote server (e.g. using FTP) or to store it into a database, use save command with a `--pipe` parameter, in connection with an external program able to store its standard input (XML) to the desired

location. You can also use similar parameter with open in order to parse documents from standard output of some external program.

Example 1. Store a XSH2 document on a remote machine using the Secure Shell

```
xsh> save --pipe "ssh my.remote.org 'cat > test.xml'" $doc
```

1.1.1. Related topics

backups	turn on backup file creation
catalog	use a catalog file during all parsing processes
switch-to-new-documents	set on/off changing current document to newly open/created files
clone	clone a given document
close	close document (without saving)
create	make a new document from a given XML fragment
document	specifying documents
filename	specifying filenames
documents	display a list of open documents
index	index a static document for faster XPath lookup
nobackups	turn off backup file creation
nodename	specifying names of DOM nodes
open	load an XML, HTML, or Docbook SGML document from a file, pipe or URI
process-xinclude	load and insert XInclude sections
save	save a document as XML or HTML
set_filename	change filename or URL associated with a document
stream	process selected elements from an XML stream (EXPERIMENTAL)
subroutine	name of a sub-routine
$variable	

1.2. Tree navigation

With XSH2, it is possible to browse a document tree (XML data represented as a DOM-tree) as if it was a local filesystem, except that XPath expressions are used instead of ordinary directory paths.

To mimic the filesystem navigation as closely as possible, XSH2 contains several commands named by analogy of UNIX filesystem commands, such as cd, ls and pwd.

The current position in the document tree is called the *current node*. Current node's XPath may be queried with pwd command. In the interactive shell, current node is also displayed in the command line prompt. (Since there may be multiple document trees open at the same time, XSH2 tries to locate a variable holding the current document and use it to fully qualify current node's XPath in the XSH2 prompt.) Remember, that beside cd command, current node (and document) is also silently changed by open command, create command and temporarily also by the node-list variant of the foreach loop without a loop variable.

XPath expressions are always evaluated in context of the current node. Different documents can be accessed through variables: `$doc/foo[1]/bar`.

Example 2. XSH2 shell

```
$scratch:/> $docA := open "testA.xml"
$docA/> $docB := open "testB.xml"
$docB/> pwd
/
$docB/> cd $docA/article/chapter[title='Conclusion']
$docA/article/chapter[5]> pwd
/article/chapter[5]
$docA/article/chapter[5]> cd previous-sibling::chapter
$docA/article/chapter[4]> cd ..
$docA/article> cd $docB
$docB:/> ls
<?xml version="1.0" encoding="utf-8"?>
<article>...</article>
```

1.2.1. Related topics

canonical	serialize nodes as to canonical XML
cd	change current context node
fold	mark elements to be folded by list command
ls	list a given part of a document as XML
locate	show a given node location (as a canonical XPath)
pwd	show current context node location (as a canonical XPath)
register-function	define XPath extension function (EXPERIMENTAL)
unfold	unfold elements folded with fold command
unregister-function	undefine extension function (EXPERIMENTAL)
xpath	XPath expression

1.3. Tree modification

XSH2 not only provides ways to browse and inspect the DOM tree but also many commands to modify its content by various operations, such as copying, moving, and deleting its nodes as well as creating completely new nodes or XML fragments and attaching them to it. It is quite easy to learn these commands since their names or aliases mimic their well-known filesystem analogies. On the other hand, many of these commands have two versions one of which is prefixed with a letter "x". This "x" stands for "cross", thus e.g. xcopy should be read as "cross copy". Let's explain the difference on the example of xcopy.

In a copy operation, you have to specify what nodes are to be copied and where to, in other words, you have to specify the *source* and the *target*. XSH2 is very much XPath-based so, XPath is used here to specify both of them. However, there might be more than one node that satisfies an XPath expression. So, the rule of thumb is that the "cross" variant of a command places *one and every* of the source nodes to the location of *one and every* destination node, while the plain variant works one-by-one, placing the first source node to the first destination, the second source node to the second destination, and so on (as long as there are both source nodes and destinations left).

```
$scratch/> $a := create "<X><A/><Y/><A/></X>";
$a/> $b := create "<X><B/><C/><B/><C/><B/></X>";
$b/> xcopy $a//A replace $b//B;
```

```
$b/> copy $b//C before $a//A;
$b/> ls $a;
<?xml version="1.0" encoding="utf-8"?>
<X><C/><A/><Y/><C/><A/></X>

$b/> ls $b;
<?xml version="1.0" encoding="utf-8"?>
<X><A/><A/><C/><A/><A/><C/><A/><A/></X>
```

As already indicated by the example, another issue of tree modification is the way in which the destination node determines the target location. Should the source node be placed before, after, or somewhere among the children of the resulting node? Or maybe, should it replace it completely? This information has to be given in the location argument that usually precedes the destination XPath.

Now, what happens if source and destination nodes are of incompatible types? XSH2 tries to avoid this by implicitly converting between node types when necessary. For example, if a text, comment, and attribute node is copied into, before or after an attribute node, the original value of the attribute is replaced, prepended or appended respectively with the textual content of the source node. Note however, that *element nodes are never converted* into text, attribute or any other textual node. There are many combinations here, so try yourself and see the results.

You may even use some more sophisticated way to convert between node types, as shown in the following example, where an element is first commented out and than again uncommented. Note, that the particular approach used for resurrecting the commented XML material works only for well-balanced chunks of XML.

Example 3. Using string variables to convert between different types of nodes

```
$doc := create <<EOF;
<?xml version='1.0'?>
<book>
  <chapter>
    <title>Intro</title>
  </chapter>
  <chapter>
    <title>Rest</title>
  </chapter>
</book>
EOF


# comment out the first chapter
ls //chapter[1] |> $chapter_xml;
insert comment $chapter_xml replace //chapter[1];
ls / 0;
# OUTPUT:
<?xml version="1.0"?>
<book>
<!--  <chapter>
    <title>Intro</title>
  </chapter>
-->
  <chapter>
    <title>Rest</title>
  </chapter>
</book>


# un-comment the chapter
$comment = string(//comment()[1]);
insert chunk $comment replace //comment()[1];
ls / 0;
# OUTPUT:
<?xml version="1.0"?>
<book>
  <chapter>
    <title>Intro</title>
  </chapter>

  <chapter>
    <title>Rest</title>
  </chapter>
</book>
```

1.3.1. Related topics

change-ns-prefix	change namespace prefix (EXPERIMENTAL)
change-ns-uri	change namespace URI (EXPERIMENTAL)
clone	clone a given document
copy	copy nodes (in the one-to-one mode)
declare-ns	create a special attribute declaring an XML namespace (EXPERIMENTAL)
edit	Edit parts of a XML document in a text editor.
edit-string	Edit a string or variable in a text editor.

expression	expression argument type
hash	index selected nodes by some key value
insert	create a node in on a given target location
location	relative destination specification (such as after, before, etc.)
map	transform node value/data using Perl or XPath expression
move	move nodes (in the one-to-one mode)
node-type	node type specification (such as element, attribute, etc.)
normalize	normalizes adjacent textnodes
process-xinclude	load and insert XInclude sections
remove	remove given nodes
rename	quickly rename nodes with in-line Perl code
set	create or modify document content (EXPERIMENTAL)
set-dtd	set document's DTD declaration
set-enc	set document's charset (encoding)
set-ns	set namespace of the current node (EXPERIMENTAL)
set-standalone	set document's standalone flag
sort	sort a given node-list by given criteria
strip-whitespace	strip leading and trailing whitespace
wrap	wrap given nodes into elements
wrap-span	wrap spans of nodes into elements
xcopy	copy nodes (in the all-to-every mode)
xinsert	create nodes on all target locations
xmove	move nodes (in the all-to-every mode)
xpath	XPath expression
xslt	compile a XSLT stylesheet and/or transform a document with XSLT
xupdate	apply XUpdate commands on a document

1.4. Flow control

As almost every scripting language, XSH2 supports subroutines, various conditional statements, loops and even exceptions.

1.4.1. Related topics

command-or-block	single XSH2 command or a block of XSH2 commands

call	indirect call to a user-defined routine (macro)
def	sub-routine declaration
do	execute a given block of commands
eval	evaluate given expression as XSH commands
exit	exit XSH2 shell
foreach	loop iterating over a node-list or a perl array
if	if statement
ifinclude	conditionally include another XSH2 source in current position
include	include another XSH2 source in current position
iterate	iterate a block over current subtree
last	immediately exit an enclosing loop
next	start the next iteration of an enclosing loop
prev	restart an iteration on a previous node
redo	restart the innermost enclosing loop block
return	return from a subroutine
run-mode	switch into normal execution mode (quit test-mode)
stream	process selected elements from an XML stream (EXPERIMENTAL)
test-mode	do not execute any command, only check the syntax
throw	throw an exception
try	try/catch statement
undef	undefine sub-routine or variable
unless	negated if statement
while	simple while loop

1.5. Retrieving more information

Beside the possibility to browse the DOM tree and list some parts of it (as described in Navigation), XSH2 provides commands to obtain other information related to open documents as well as the XSH2 interpreter itself. These commands are listed bellow.

1.5.1. Related topics

apropos	search the documentation
canonical	serialize nodes as to canonical XML
count	calculate a expression and enumerate node-lists
doc-info	displays various information about a document

get	calculate a given expression and return the result.
documents	display a list of open documents
help	on-line documentation
lineno	print line-numbers corresponding to matching nodes
ls	list a given part of a document as XML
defs	list all user-defined subroutines
dtd	show document's DTD
locate	show a given node location (as a canonical XPath)
namespaces	List namespaces available in a context of a given nodes
settings	list current settings using XSH2 syntax
print	print stuff on standard or standard error output
enc	show document's original character encoding
pwd	show current context node location (as a canonical XPath)
validate	validate a document against a DTD, RelaxNG, or XSD schemas
variables	list global variables
version	show version information

1.6. Namespaces in XML and XPath

Namespaces provide a simple method for qualifying element and attribute names in XML documents. Namespaces are represented by a namespace URI but, since the URI can be very long, element and attribute names are associated with a namespace using a namespace prefix (see the W3C recommendation [http://www.w3.org/TR/REC-xml-names/] for details). In an XML document, a prefix can be associated with a namespace URI using a declaration which takes form of special attribute of the form `xmlns:prefix="namespace uri"` on an element. The scope of the namespace declaration is then the subtree of the element carrying the special `xmlns:prefix` attribute (and includes attributes of the element). Moreover, a default namespace can be declared using just `xmlns="namespace uri"`. In that case all unprefixed element names in the scope of such a declaration belong to the namespace. An unprefixed element which is not in scope of a default namespace declaration does not belong to any namespace. It is recommended not to combine namespaced elements and non-namespaced elements in a single document. Note that regardless of default namespace declarations, unprefixed attributes do not belong to any namespace (because they are uniquely determined by their name and the namespace and name of the the element which carries them).

XSH2 tries to deal namespace declarations transparently (creating them if necessary when nodes are copied between different documents or scopes of namespace declarations). Most commands which create new elements or attributes provide means to indicate a namespace. In addition, XSH2 provides commands declare-ns, set-ns, change-ns-uri, and change-ns-prefix to directly manipulate XML namespace declarations on the current node.

Since XSH2 is heavily XPath-based, it is important to remember that XPath 1.0 maps prefixes to namespaces independently of the declarations in the current document. The mapping is instead provided via so called XPath context. Namespaces can be tested in XPath either using the built-in `namespace-uri()` function, but it is more convenient to use namespace prefixes associated with namespace URIs in the XPath context. This association is independent of the documents to which the XPath expression is applied and can be established using the command register-namespace. Additional, XSH2 automatically propagates the namespace association in the scope of the current node to the XPath context, so that per-document prefixes in the current scope can also be used.

IMPORTANT: XPath 1.0 [http://www.w3.org/TR/xpath] has no concept of a default namespace. Unprefixed names in XPath only match names which have no namespace. So, if the document uses a default namespace, it is required to associate a non-empty prefix with the default namespace via register-namespace and add that prefix to names in XPath expressions intended to match nodes in the default namespace.

Example 4. Manipulating nodes in XHTML documents

```
open "index.xhtml";
$xhtmlns = "http://www.w3.org/1999/xhtml";
register-namespace x $xhtmlns;
wrap --namespace $xhtmlns '<font color="blue">' //x:a[@href];
# or
wrap '<x:font color="blue">' //x:a[@href];
```

In the preceding example we associate the (typically default) namespace of XHTML documents with the prefix `x`. We than use this prefix to match all links (`a` elements) in the document. Note that we do not write `@x:href` to match the `@href` attribute because unprefixed attributes do not belong to the default namespace. The wrap command is used to create new containing elements for the nodes matched by the XPath expression. We may either specify the namespace of the containing element explicitly, using `--namespace` option, or implicitly, by using a prefix associated with the namespace in the XPath context. In the latter case, XSH2 chooses a suitable prefix declared for the namespace in the document scope (in this case the default, i.e. no, prefix), adding a new namespace declaration if necessary.

1.6.1. Related topics

change-ns-prefix	change namespace prefix (EXPERIMENTAL)
change-ns-uri	change namespace URI (EXPERIMENTAL)
declare-ns	create a special attribute declaring an XML namespace (EXPERIMENTAL)
namespaces	List namespaces available in a context of a given nodes
register-namespace	register namespace prefix to use XPath expressions
register-xhtml-namespace	register a prefix for the XHTML namespace
register-xsh-namespace	register a prefix for the XSH2 namespace
set-ns	set namespace of the current node (EXPERIMENTAL)
unregister-namespace	unregister namespace prefix

1.7. Argument Types

XSH2 commands accept arguments of various types, usually expressed as Perl or XPath expressions. Unlike in most languages, individual XSH2 commands may evaluate the same expression differently, usually to enforce a result of a certain type (such as a node-list, a string, a number, a filename, a node name, etc.). See expression and individual argument types for more information.

1.7.1. Related topics

command-or-block	single XSH2 command or a block of XSH2 commands
command	List of XSH2 commands and their general syntax
document	specifying documents
encoding	character encoding (codepage) identifier

expression	expression argument type
filename	specifying filenames
location	relative destination specification (such as after, before, etc.)
nodename	specifying names of DOM nodes
node-type	node type specification (such as element, attribute, etc.)
perl-code	in-line code in Perl programming language
subroutine	name of a sub-routine
$variable	
xpath	XPath expression

1.8. Variables

In XSH2, like in Perl and XPath, variable names are are prefixed with a dollar sign ($). Variables can contain arbitrary Perl Scalar (string, number, array reference, hash reference or an object reference). XPath objects are transparently mapped to Perl objects via XML::LibXML objects. Values can be assigned to variables either by simple assignments of the form `$variable = expression`, where the right hand side is an expression, or by command assignments of the form `$variable := command`, where the right hand side is a XSH2 command, or by capturing the output of some command with a variable redirection of the following form:

```
command |> $variable;
```

XSH2 expressions are evaluated either by XPath engine or by Perl (the latter only happens if the entire expression is enclosed with braces `{...}`), and both Perl and XPath can access all XSH2 variables transparently (Perl expressions may even assign to them).

A simple simple expression consisting of a variable name (e.g. `$variable`) is always evaluated by the XPath engine and the result is the content of the variable as it appears to the XPath data model. Since in XPath object cannot be void (undefined), XPath engine complains, if the value of the variable is undefined. On the other hand, expressions like `{$variable}` are evaluated by Perl, which results in the value of the variable as seen by Perl.

Variables can also be used as macros for complicated XPath expressions. Any occurrence of a substring of the form `${variable}` in an XPath expression is interpolated to the value of `$variable` (if `$variable` contains an object rather than a string or number, then the object is cast to string first) before the entire expression is evaluated. So, for example, if `${variable}` contains string "`chapter[title]`" (without the quotes), then the XPath expression `//sect1/${variable}/para` interpolates to `//sect1/chapter[title]/para` prior to evaluation.

To display the current value of a variable, use either print or (in case of a global variables - the distinction is discussed below) the command variables:

```
xsh> $b="my_document";
xsh> $file="${b}s.xml";
xsh> $f := open $file;
xsh> ls //$b[count(descendant::para)>10]
xsh> print $b
my_document
xsh> variables
...
$b='my_document';
...
$file='my_documents.xml';
...
```

Variables can also serve as containers for documents and can be used to store lists of nodes that result from evaluating an XPath expression (a.k.a. XPath node-sets). This is especially useful when a sequence of commands is to be performed on some fixed set of nodes and repetitive evaluation of the same XPath expression would be lengthy. XPath node-sets are represented by `XML::LibXML::NodeList` Perl objects (which is simply a array reference blessed to the above class, which provides some simple operator overloading). In XPath, by a node-set by definition can only contain a single copy of each node and the nodes within a node-set are processed in the same order as they appear in the XML document. Having XPath node-sets represented by a list gives us the advantage of having the possibility to process the list in a different order than the one implied by the document (which is what happens if a variable containing a node-list is evaluated by Perl rather than XPath), see an example below.

```
xsh> $creatures = //creature[@status='alive']
# process creatures in the document order:
xsh> foreach $creature print @name;
# process creatures in the reverse document order:
xsh> foreach { reverse @$creature } print @name;
# append some more nodes to a node-list (using a variant of
# a simple assignment)
xsh> $creatures += //creature[@status='dead'];
# again, we can process creatures in order implied by the document:
xsh> foreach $creature print @name;
# but we can also process first living and then dead creatures,
# since this is how they are listed in $creature
xsh> foreach {$creature} print @name;
# same as the above is
xsh> foreach {@$creature} print @name;
```

XSH2 variables are either globally or lexically scoped. Global variables need not to be declared (they can be directly assigned to), whereas lexical variables must be declared using the command my. Global variable assignment may also be made temporal for the enclosing block, using local.

```
$var1 = "foo";           # a global variable requires no declaration
local $var1 $var2 $var3; # localizes global variables
$var1 = "bar";           # assignment to a localized variable is temporary
local $var4 = "foo";     # localized assignment
my $var1 $var $var3;     # declares lexical variables
my $var1 = "foo";        # lexical variable declaration with assignment
```

Lexical variables are only defined in the scope of current block or subroutine. There is no way to refer to a lexical variable form outside of the block it was declared in, nor from within a nested subroutine call. Of course, lexical variables can be referred to from nested blocks or Perl expressions (where they behave just like Perl's lexical variables).

On the other hand, global or localized XSH2 variables are just Perl Scalar variables belonging to the `XML::XSH2::Map` namespace, which is also the default namespace for any Perl code evaluated from XSH2 (so there's no need to use this prefix explicitly in Perl expressions, unless of course there is a lexical variable in the current scope with the same).

Localizing a variable using the `local` keyword makes all assignments to it occurring in the enclosing block temporary. The variable itself remains global, only its original value is restored at the end of the block that localized it.

In all above cases, it is possible to arbitrarily intermix XSH2 and Perl assignments:

```
xsh> ls //chapter[1]/title
<title>Introduction</title>
xsh> $a=string(//chapter[1]/title)
xsh> perl { $b="CHAPTER 1: ".uc($a); }
xsh> print $b
CHAPTER 1: INTRODUCTION
```

Although all XSH2 variables are in fact Perl Scalars, it is still possible to store Perl Array or Hash value to a XSH2 variable via reference. The following example demonstrates using Perl Hashes to collect and print some simple racial statistics about the population of Middle-Earth:

```
my $races;
foreach a:/middle-earth/creature {
  my $race=string(@race);
  perl { $races->{$race}++ };
}
print "Middle-Earth Population (race/number of creatures)"
print { map "$_/$races->{$_}\n" keys(%$races); };
```

1.8.1. Related topics

assign variable assignment

document specifying documents

expression expression argument type

local temporarily assign new value to a variable

subroutine name of a sub-routine

$variable

xpath XPath expression

1.9. Command output redirection

WARNING: XSH2 redirection syntax is not yet finished. It is currently the same as in XSH1 but this may be drastically changed in the future releases.

Output redirection can be used to pipe output of some XSH command to some external program, or to capture it to a variable. Redirection of output of more than one XSH command can be achieved using the do command.

1.9.1. Redirect output to an external program

The syntax for redirecting the output of a XSH command to an external program, is `xsh-command | shell-command ;`, where `xsh-command` is any XSH2 command and `shell-command` is any command (or code) recognized by the default shell interpreter of the operating system (i.e. on UNIX systems by `/bin/sh` or `/bin/csh`, on Windows systems by `cmd`). The shell command may contain further redirections (as supported by the system shell interpreter), but should not contain semicolons, except when the whole shell command is enclosed in brackets.

Example 5. Use well-known UNIX commands to filter XPath-based XML listing from a document and count the results

```
xsh> ls //something/* | grep foo | wc
```

1.9.2. Capture output to a variable

The syntax for capturing the output of an XSH command to a variable is `xsh-command |> $variable`, where `xsh-command` is any XSH command and `$variable` is any valid name for a variable.

Example 6. Store the number of all words in a variable named count.

```
xsh> count //words |> $count
```

1.10. Global settings

The commands listed below can be used to modify the default behavior of the XML parser or XSH2 itself. Some of the commands switch between two different modes according to a given expression (which is expected to result either in zero or non-zero value). Other commands also working as a flip-flop have their own explicit counterparts (e.g. verbose and quiet or debug and nodebug). This inconsistency is due to historical reasons.

The encoding and query-encoding settings allow to specify character encodings of user's input and XSH2's own output. This is particularly useful when you work with UTF-8 encoded documents on a console which only supports 8-bit characters.

The settings command displays current settings by means of XSH2 commands. Thus it can not only be used to review current values, but also to store them for future use, e.g. in ~/.xshrc file.

```
xsh> settings | cat > ~/.xshrc
```

1.10.1. Related topics

backups	turn on backup file creation
switch-to-new-documents	set on/off changing current document to newly open/created files
parser-completes-attributes	turn on/off parser's ability to fill default attribute values
debug	display many annoying debugging messages
empty-tags	turn on/off serialization of empty tags
encoding	character encoding (codepage) identifier
encoding	choose output charset
indent	turn on/off pretty-printing
keep-blanks	turn on/off ignorable whitespace preservation
load-ext-dtd	turn on/off external DTD fetching
nobackups	turn off backup file creation
nodebug	turn off debugging messages
settings	list current settings using XSH2 syntax
parser-expands-entities	turn on/off parser's tendency to expand entities
parser-expands-xinclude	turn on/off transparent XInclude insertion by parser
pedantic-parser	make the parser more pedantic
query-encoding	declare the charset of XSH2 source files and terminal input
quiet	turn off many XSH2 messages
recovering	turn on/off parser's ability to fix broken XML
register-function	define XPath extension function (EXPERIMENTAL)
register-namespace	register namespace prefix to use XPath expressions
register-xhtml-namespace	register a prefix for the XHTML namespace

register-xsh-namespace	register a prefix for the XSH2 namespace
run-mode	switch into normal execution mode (quit test-mode)
skip-dtd	turn on/off serialization of DTD DOCTYPE declaration
test-mode	do not execute any command, only check the syntax
unregister-function	undefine extension function (EXPERIMENTAL)
unregister-namespace	unregister namespace prefix
validation	turn on/off validation in XML parser
verbose	make XSH2 print many messages
xpath-axis-completion	sets TAB completion for axes in xpath expressions in the interactive mode
xpath-completion	turn on/off TAB completion for xpath expressions in the interactive mode
xpath-extensions	map predefined XSH2 XPath extension functions to no or other namespace

1.11. Interacting with Perl and Shell

Along with XPath, Perl is one of two XSH2 expression languages, and borrows XSH2 its great expressive power. Perl is a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It has built-in regular expressions and powerful yet easy to learn data structures (scalars, arrays, hash tables). It's also a good language for many system management tasks. XSH2 itself is written in Perl (except for the XML engine, which uses libxml2 library written in C by Daniel Veillard).

1.11.1. Calling Perl

Perl expressions or blocks of code can either be used as arguments to any XSH2 command. One of them is perl command which simply evaluates the given Perl block. Other commands, such as map, even require Perl expression argument and allow quickly change DOM node content. Perl expressions may also provide lists of strings to iterate over with a foreach loop, or serve as conditions for if, unless, and while statements.

To prevent conflict between XSH2 internals and the evaluated Perl code, XSH2 runs such code in the context of a special namespace `XML::XSH2::Map`. As described in the section Variables, XSH2 string variables may be accessed and possibly assigned from Perl code in the most obvious way, since they actually *are* Perl variables defined in the `XML::XSH2::Map` namespace.

The interaction between XSH2 and Perl actually works the other way round as well, so that you may call back XSH2 from the evaluated Perl code. For this, Perl function `xsh` is defined in the `XML::XSH2::Map` namespace. All parameters passed to this function are interpreted as XSH2 commands.

Moreover, the following Perl helper functions are defined:

`xsh(string,....)` - evaluates given string(s) as XSH2 commands.

`call(name)` - call a given XSH2 subroutine.

`count(string)` - evaluates given string as an XPath expression and returns either literal value of the result (in case of boolean, string and float result type) or number of nodes in a returned node-set.

`literal(string|object)` - if passed a string, evaluates it as a XSH2 expression and returns the literal value of the result; if passed an object, returns literal value of the object. For example, `literal('$doc/expression')` returns the same value as `count('string($doc/expression)')`.

`serialize(string|object)` - if passed a string, it first evaluates the string as a XSH2 expression to obtain a node-list object. Then it serializes the object into XML. The resulting string is equal to the output of the XSH2 command ls applied on the same expression or object expression only without indentation and folding.

`type(string|object)` - if passed a string, it first evaluates the string as XSH2 expression to obtain a node-list object. It returns a list of strings representing the types of nodes in the node-list (ordered in the canonical document order). The returned type strings are: `element`, `attribute`, `text`, `cdata`, `pi`, `entity_reference`, `document`, `chunk`, `comment`, `namespace`, `unknown`.

`nodelist(string|object,...)` - converts its arguments to objects if necessary and returns a node-list consisting of the objects.

`xpath(string, node?)` - evaluates a given string as an XPath expression in the context of a given node and returns the result.

`echo(string,...)` - prints given strings on XSH2 output. Note, that in the interactive mode, XSH2 redirects all output to a specific terminal file handle stored in the variable `$OUT`. So, if you for example mean to pipe the result to a shell command, you should avoid using STDOUT filehandle directly. You may either use the usual `print` without a filehandle, use the `echo` function, or use `$OUT` as a filehandle.

In the following examples we use Perl to populate the Middle-Earth with Hobbits whose names are read from a text file called `hobbits.txt`, unless there are some Hobbits in Middle-Earth already.

Example 7. Use Perl to read text files

```
unless (//creature[@race='hobbit']) {
  perl {
    open my $fh, "hobbits.txt" };
    @hobbits=<$file>;
    close $fh;
  }
  foreach { @hobbits } {
    copy xsh:new-element("creature","name",.,"race","hobbit")
      into m:/middle-earth/creatures;
  }
}
```

Example 8. The same code as a single Perl block

```
perl {
  unless (count(//creature[@race='hobbit'])) {
    open my $file, "hobbits.txt";
    foreach (<$file>) {
      xsh(qq{insert element "<creature name='$_' race='hobbit'>"
        into m:/middle-earth/creatures});
    }
    close $file;
  }
};
```

1.11.2. Writing your own XPath extension functions in Perl

XSH2 allows users to extend the set of XPath functions by providing extension functions written in Perl. This can be achieved using the register-function command. The perl code implementing an extension function works as a usual perl routine accepting its arguments in `@_` and returning the result. The following conventions are used:

The arguments passed to the perl implementation by the XPath engine are simple scalars for string, boolean and float argument types and `XML::LibXML::NodeList` objects for node-set argument types. The implementation is responsible for checking the argument number and types. The implementation may use general Perl functions as well as `XML::LibXML` methods to process the arguments and return the result. Documentation for the `XML::LibXML` Perl module can be found for example at http://search.cpan.org/~pajas/XML-LibXML/.

Extension functions SHOULD NOT MODIFY the document DOM tree. Doing so could not only confuse the XPath engine but possibly even result in an critical error (such as segmentation fault). Calling XSH2 commands from extension function implementations is also dangerous and isn't generally recommended.

The extension function implementation must return a single value, which can be of one of the following types: simple scalar (a number or string), `XML::LibXML::Boolean` object reference (result is a boolean value), `XML::LibXML::Literal` object reference (result is a string), `XML::LibXML::Number` object reference (result is a float), `XML::LibXML::Node` (or derived) object reference (result is a node-set consisting of a single node), or `XML::LibXML::NodeList` (result is a node-set). For convenience, simple (non-blessed) array references consisting of `XML::LibXML::Node` objects can also be used for a node-set result instead of a `XML::LibXML::NodeList`.

1.11.3. Calling the System Shell

In the interactive mode, XSH2 interprets all lines starting with the exclamation mark (`!`) as shell commands and invokes the system shell to interpret the line (this is to mimic FTP and similar command-line interpreters).

```
xsh> !ls -l
-rw-rw-r--    1 pajas    pajas        6355 Mar 14 17:08 Artistic
drwxrwxr-x    2 pajas    users         128 Sep  1 10:09 CVS
-rw-r--r--    1 pajas    pajas       14859 Aug 26 15:19 ChangeLog
-rw-r--r--    1 pajas    pajas        2220 Mar 14 17:03 INSTALL
-rw-r--r--    1 pajas    pajas       18009 Jul 15 17:35 LICENSE
-rw-rw-r--    1 pajas    pajas         417 May  9 15:16 MANIFEST
-rw-rw-r--    1 pajas    pajas         126 May  9 15:16 MANIFEST.SKIP
-rw-r--r--    1 pajas    pajas       20424 Sep  1 11:04 Makefile
-rw-r--r--    1 pajas    pajas         914 Aug 26 14:32 Makefile.PL
-rw-r--r--    1 pajas    pajas        1910 Mar 14 17:17 README
-rw-r--r--    1 pajas    pajas         438 Aug 27 13:51 TODO
drwxrwxr-x    5 pajas    users         120 Jun 15 10:35 blib
drwxrwxr-x    3 pajas    users        1160 Sep  1 10:09 examples
drwxrwxr-x    4 pajas    users          96 Jun 15 10:35 lib
-rw-rw-r--    1 pajas    pajas           0 Sep  1 16:23 pm_to_blib
drwxrwxr-x    4 pajas    users         584 Sep  1 21:18 src
drwxrwxr-x    3 pajas    users         136 Sep  1 10:09 t
-rw-rw-r--    1 pajas    pajas          50 Jun 16 00:06 test
drwxrwxr-x    3 pajas    users         496 Sep  1 20:18 tools
-rwxr-xr-x    1 pajas    pajas        5104 Aug 30 17:08 xsh
```

To invoke a system shell command or program from the non-interactive mode or from a complex XSH2 construction, use the exec command.

Since UNIX shell commands are very powerful tool for processing textual data, XSH2 supports direct redirection of XSH2 commands output to system shell command. This is very similarly to the redirection known from UNIX shells, except that here, of course, the first command in the pipe-line colone is an XSH2 command. Since semicolon (`;`) is used in XSH2 to separate commands, it has to be prefixed with a backslash if it should be used for other purposes.

Example 9. Use grep and less to display context of `funny'

```
xsh> ls //chapter[5]/para | grep funny | less
```

Example 10. The same on Windows 2000/XP systems

```
xsh> ls //chapter[5]/para | find "funny" | more
```

1.11.4. Related topics

lcd	change system working directory
exec	execute a shell command
expression	expression argument type
hash	index selected nodes by some key value
map	transform node value/data using Perl or XPath expression
perl-code	in-line code in Perl programming language
perl	evaluate in-line Perl code
rename	quickly rename nodes with in-line Perl code

1.12. Prompt in the interactive shell

Like many other shells, XSH2 provides means for customizing the format of its interactive shell prompt. The prompt is displayed according to the content of the variable `$PROMPT` on which the following substitutions and interpolations are performed (in this order):

1. Prompt-string replacements

```
%% - percent sign
%p - XPath location of the current node
%P - like %p but without an initial document variable
%l - XPath location of the current node with ID-shortcuts
%L - like %l but without an initial document variable
%n - name of the current node
%N - local name of the current node
%c - canonical XPath name of the current node
%y - type of the current node (element,attribute,...)
%i - ID of the current node
%d - current document variable
%h - the hostname up to the first '.'
%H - the hostname
%s - XSH shell name (basename of $0)
%t - the current time in 24-hour HH:MM:SS format
%T - the current time in 12-hour HH:MM:SS format
%@ - the current time in 12-hour am/pm format
%A - the current time in 24-hour HH:MM format
%u - the username of the current user
%v - the version of XSH2 (e.g., 2.1.0)
%V - the revision number of XML::XSH2::Functions (e.g. 2.40)
%w - current working directory (on the local filesystem)
%W - basename of %w
```

2. Variable, XPath and Perl interpolations

Substrings of the forms `${variable}`, `${{...perl...}}` and `${(...xpath...)}` are interpolated as in XSH2 expressions.

3. Special character substitution

```
\n - newline character
\r - line-feed character
\t - tab character
\a - bell character
\b - backspace character
\f - form feed character
\e - escape character (\033)
\\ - backslash character
\nnn - the character corresponding to the octal number nnn
(useful for non-printable terminal control characters)
```

The default value of `$PROMPT` is `"%p>"`.

Note that you must escape `${...}` interpolators like `\${...}` if you want them to be evaluated at each prompt rather than at the time of the assignment to `$PROMPT`. For example:

Example 11. Let `uname` be computed once, `date` at every prompt

```
$PROMPT="[${{ chomp($u=`uname`);$u }} \${{ chomp($d=`date`);$d }}] %p>"
```

1.13. Changes since XSH 1.x

This section briefly describes differences between XSH2 and previous XSH 1.x releases. The list should not be considered complete. Some syntax variations or amendments in the semantics of various commands may not be documented in this section, neither are various improvements in the XSH interpreter.

1.13.1. Changes in XSH2

1. In XSH2, subroutines can be called without a call. They can be redefined and undefined. The command call can still be used, but it's use only makes sense in indirect calls, where subroutine's name is computed from an expression.

   ```
   def foo $param1 $param2 {
     # param1 and $param2 are lexical (a.k.a. my)
     ls $param1;
     echo $param2
   }
   foo (//chapter)[1] (//chapter)[1]/title

   def inc $param1 { return ($param1 + 1) }
   $two := inc 1;
   ```

2. XSH2 uses variables of the form $variable for all kinds of objects, including node-sets (which, if evaluated as Perl expressions, preserve node order). Node-list variables of XSH 1.x have been deprecated.

   ```
   $var = //foo/bar;                 # node set
   $var = "hallo world";             # string
   $var = xsh:new-element("foo");    # node object
   $var = { ['a','b','c'] };         # Perl array reference
   $var = {{ 'a'=>'A', 'b'=>'B' }};  # Perl hash reference
   ```

3. XSH2 allows variables to be used in XPath just as they are used in XSLT:

   ```
   $var = //foo/bar;
   ls //baz[ . = $var[@test=1]/any ]
   ```

   Variable interpolation is still available in XSH2 via ${var}, but it's importance is diminished compared to XSH 1.x, because the XPath engine now evaluates variables directly. Interpolation can still be used for things like "XPath-macros":

```
$filter = "[ . = $var[@test=1]/any ]";
ls //baz${filter};
```

4. XSH2 equally supports XPath and Perl expressions (written in braces { ... }). Unfortunately, Perl expressions can't be embedded in XPath expressions, but one can still use variables as an agent:

```
perl { use MIME::Base64 };
my $encoded = { encode_base64('open sesame') }
ls //secret-cave[string(password) = $encoded]
```

   We can, however, use Perl-only expressions complemented with auto-conversion to do things like:

```
copy { encode_base64('Pe do mellon a minno!') } replace
//secret-cave/password/text();
```

5. Commands return values (see := assignment, or &{ } expressions).

```
$moved_paras := xmove //para replace .;
$chapter := wrap chapter $moved_paras;
ls $chapter;

# or just

ls &{ wrap chapter &{ xmove //para replace . } };
```

6. XSH2 deprecates "string" expressions of XSH 1.x. However, for convenience, some XSH2 commands interpret name-like XPath expressions on certain argument positions as strings (mostly commands that expect file-name or node-name arguments):

```
insert element my_document into .;
insert text "foo" into my_document;

$doc := open my_document;         # opens file named "my_document"
$doc := open "my_document";       # same
$doc := open (my_document);       # opens file named "foo"
$doc := open string(my_document); # same
```

7. In XSH2, XML documents have no ID. They are referred to using variables (which fits in well with the unified variable concept):

```
$doc1 := open "foo1.xml";
$doc2 := open "foo2.xml";
ls ($doc1//para|$doc2//para);
cd $doc1;
ls id('intro');             # finds ID intro in the current document ($doc1)
ls xsh:id2($doc2, 'intro'); # finds ID intro in $doc2
```

8. XSH2 commands have options and flags instead of many optional (positional) arguments. Options/flags usually have both long forms (like --flag) and equivalent short forms (like :f) (colon is borrowed from Scheme, because dash is reserved for XPath minus).

```
$doc := open --format html "version1.html";
save --file "version2.xml" $doc;

ls --fold /;
ls :f /;
ls --depth 1 /;
ls :d 1 /;

# all the same:
```

```
$sorted = sort --key @name --locale --descending //user;
$sorted = sort :l:d:k@name //user;
$sorted = sort --key @name --compare { use locale; $b cmp $a } //user;

validate --relaxng --file "test.rng" $mydoc;
validate --public "-//OASIS//DTD DocBook XML V4.1.2//EN" $mydoc;
validate --yesno $mydoc;
```

9. Finally, eval is no longer an alias for perl in XSH2, but instead evaluates strings containing XSH2 commands (so `eval $string` now practically works like old ugly `perl { xsh($string) }`). See the documentation for eval for a handy usage example (no more PHP, XSTL and XPathScript :-)).

1.13.2. Examples

Example 12. Open command has changed.

```
XSH1:
foo = file.xml;
or
foo = "file.xml";
```

```
XSH2:
$foo := open file.xml;        # file.xml is a bareword in file-name context
or
$foo := open "file.xml";      # "file.xml" is a XPath string
or
$foo := open {"file.xml"};    # "file.xml" is a Perl string
or
$foo = xsh:open("file.xml");  # righthand side is an XPath extension function
```

Example 13. XSH2 commands have options

```
XSH1:
open HTML FILE foo2 = "file.html";
```

```
XSH2:
$foo2 := open --format html "file.html";
```

Example 14. documents

```
XSH1:
foo = file.xml;
ls foo:(//bar|//baz);
```

```
XSH2:
$foo := open file.xml;
ls ($foo//bar|$foo//baz);
```

Example 15. variable interpretation

```
XSH1:
$family = "Arial";
ls //font[@family="$family"];   # interpolation
or
ls //font[@family="${family}"]; # interpolation
```

```
XSH2:
$family = "Arial";
ls //font[@family=$family];     # evaluation by XPath engine
or
ls //font[@family="${family}"]; # interpolation
```

Example 16. adding new nodes

```
XSH1:
insert attribute "foo=bar" into /scratch;
```

```
XSH2:
insert attribute "foo=bar" into /scratch;
or
copy xsh:new-attribute("foo","bar") into /scratch;
```

Example 17. foreach with perl expression

```
XSH1:
foreach { glob('*.xml') } {
  open doc = $__;
  ...
}
```

```
XSH2:
foreach { glob('*.xml') } {
  my $doc := open .;
  ...
}
```

Example 18. foreach (perl expression) with variable

```
XSH2:
foreach my $filename in { glob('*.xml') } {
  my $doc := open $filename;
  ...
}
```

Example 19. sorting nodes

```
XSH1:
%list = //player;
sort @best_score { $a <=> $b } %list;
copy %list into .;
```

```
XSH2:
$list := sort --numeric --key @best_score //player;
copy { $list } into .;
or
copy &{ sort --numeric --key @best_score //player } into .;
or (using short options)
copy &{ sort :n :k @best_score //player } into .;
```

2. Command Reference

2.1. apropos

Usage

```
apropos [--fulltext] [--regexp] expression
```

Description

Print all help topics containing given expression in their short description. The `--fulltext` flag forces the search to be performed over the full text of help. `--fulltext` indicates, that the given expression is a regular expression instead of a literal string.

2.2. assign

Usage

```
[assign] $variable = expression[assign] $variable := command[assign] $variable
[-= | += | *= | /= | %= | x= | .= | ||= | &&= ] expression[assign] $variable
[-:= | +:= | *:= | /:= | %:= | x:= | .:= | ||:= | &&:= ] command
```

Description

Evaluate the expression (= assignment) or command (:= assignment) on the right side of the assignment and store the result in a given variable. Optionally a Perl operator (- subtraction, + addition, * multiplication, / division, % modulo, x repeat string n-times, . concatenation, || logical OR, && logical AND) can precede the assignment, in which case the variable is assigned the result of applying given operator on its previous value and the value of the right side of the assignment.

Example 20. Assign XPath (node-set, string), or Perl results

```
xsh> $a=chapter/title;
xsh> $b="hallo world";
xsh> $c={ `uname` };
xsh> ls $a;
```

Example 21. Arithmetic expressions (XPath)

```
xsh> $a=5*100 # assign 500 to $a
xsh> $a += 20 # add 20 to $a
xsh> $a = (($a+5) div 10)
```

Example 22. Arithmetic expressions (Perl)

```
xsh> $a={ 5*100 }
xsh> $a = { join ';', split //,"hallo" } # assigns "h;a;l;l;o" to $a
```

Example 23. Command result assignment

```
xsh> $doc := open "file.xml" # open a document
xsh> $copies := xcopy //foo into //bar # copy elements and store the copies
xsh> $wrappers := wrap "wrapper" $copies # wrap each node from $copies to a new element
 "wrapper" and store the wrapping elements
```

See Also

variables

2.3. backups

Usage

```
backups
```

Description

Enable creating backup files on save (default).

This command is equivalent to setting the `$BACKUPS` variable to 1.

See Also

nobackups

2.4. call

Usage

```
call expression [expression ...]
```

Description

Call a subroutine whose name is computed by evaluating the first argument expression. All other expressions are evaluated too and the results are passed to the subroutine as arguments.

This command should only be used if the name of the subroutine isn't known at the compile time. Otherwise it is recommended to use a direct subroutine call of the form:

```
subroutine-name [argument1 argument2 ...]
```

```
def a $arg { echo "A says" $arg }
def b $arg { echo "B says" $arg }
a "hallo!";  # call subroutine a
b "hallo!";  # call subroutine b
call { chr(ord("a")+rand(2)) } "surprise!"; # call a or b randomly
```

See Also

def, return

2.5. canonical

Usage

```
canonical [--comments|:c] [--filter|:f xpath] [expression]
```

Description

This commands prints a canonical XML representing nodes specified by its argument (or current node, if no argument is given).

`--comments` or `:c` removes comments from the resulting XML.

`--filter` or `:f` can be used to filter the resulting XML so that it only contains nodes explicitly included in the given node-set.

For details see "Canonical XML" [http://www.w3.org/TR/xml-c14n] or "Exclusive XML Canonicalization" [http://www.w3.org/TR/xml-exc-c14n] W3C recommendations.

See Also

ls

2.6. catalog

Usage

```
catalog filename
```

Description

Make use of a given XML file as a catalog during all parsing processes. Using a catalog may significantly speed up parsing processes if many external resources are loaded into the parsed documents (such as DTDs or XIncludes).

2.7. cd

Usage

```
cd [expression]
```

Aliases

```
chxpath
```

Description

Evaluate given expression to a node-list and change current context node to the first node in it.

2.8. change-ns-prefix

Usage

```
change-ns-prefix expression [expression]
```

Description

This command takes one or two arguments. The first argument is a new prefix and the second, optional, argument is an old namespace prefix. It changes the prefix of a namespace declaration of the context node to the new value. If no old prefix is given, the change applies to a declaration on the context node whose prefix equals to the prefix of the context node, otherwise the command changes the declaration with the given old prefix.

The command throws an exception if the new prefix is already taken by some other declaration in the scope.

See Also

change-ns-uri, set-ns, declare-ns, namespaces

2.9. change-ns-uri

Usage

```
change-ns-uri expression [expression]
```

Description

This command takes one or two arguments. The first argument is a new namespace URI and the second, optional, argument is a namespace prefix. It changes the URI value of a namespace declaration of the context node to the

new value. If no prefix is given, the change applies to a declaration on the context node whose prefix equals to the prefix of the context node, otherwise the change applies to a declaration with the given prefix.

See Also

change-ns-prefix, set-ns, declare-ns, namespaces

2.10. clone

Usage

```
$doc := clone document
```

Aliases

```
dup
```

Description

Create and return a copy of a given document. Unless switch-to-new-documents configuration flag is turned off, the root node of the new document becomes the current node.

Calling this command only makes sense if either switch-to-new-documents is set, or if the result is assigned to a variable or passed to another XSH2 command using the `&{...}` syntax, since otherwise the newly created copy of the document is automatically garbage-collected and destroyed.

See Also

open, close, enc, documents

2.11. close

Usage

```
close [document]
```

Description

Close a given document (or, if called with no argument, the current document) by trying to remove all references from XSH2 variables to nodes belonging to the document. If no references to the document are left, the garbage-collector destroys the DOM tree and frees the memory it occupied for later reuse (depending on architecture, this may or may not give the allocated memory back to the system).

2.12. copy

Usage

```
copy [--respective|:r] expression location expression$results := copy [--re-
spective|:r] expression location expression
```

Aliases

```
cp
```

Description

Copies nodes in the first node-list expression (source nodes) to the destinations determined by the the location directive applied to nodes in the second node-list expression (target nodes). If the source node-list contains more than one node, than N'th node in the source node-list is copied to the location relative to the N'th node in the target node-list.

If `--respective|:r` option is used, then the target node-list expression is evaluated in the context of the source node being copied.

Possible values for location are: `after`, `before`, `into`, `replace`, `append` and `prepend`. The first three location directives cause making a copy of the source nodes after, before, and within (as the last child-node) the target nodes, respectively. If `replace` location directive is used, source node are copied before the respective target nodes and target nodes are removed. The `append` and `prepend` location directives allow, depending on the destination node type, either inserting copies of the source nodes as the first or last child nodes of a destination element or appending/prepending destination node data in case of non-element destination nodes. See location argument type for more detail.

The command returns a node-list consisting of the copies of all source nodes created by the command.

Despite the fact the command is named "copy", nodes resulting from copying the source nodes may pass through certain type conversion before they are inserted at the appointed destinations. This, however, only happens in cases where the types of the source and target nodes are not compatible with the location directive. See location argument type for more detail.

Note that XSH2 refuses to create multiple top-level elements using `copy`, move and similar commands.

Example 24. Replace living-thing elements in the document b with copies of the corresponding creature elements from the document $a.

```
xsh> copy $a//creature replace $b//living-thing
```

Example 25. Copy every element into itself

```
xsh> copy --respective $a//* into .
```

```
xsh> copy $a//* into $a//*#same as
above
```

See Also

xcopy, move, xmove, insert, xinsert

2.13. count

Usage

```
count [--quiet|:q] xpath
```

Description

Calculates a given expression expression. If the result is a node-list, print number of nodes in the node-list. If the expression results in a boolean, numeric or literal value, print the value.

If `--quiet` or `:q` option is used, output is suppressed and the value is returned.

See Also

get

2.14. create

Usage

```
$doc := create nodename|expression
```

Aliases

```
new
```

Description

Returns a new document object. The argument must evaluate either to a valid element name (possibly followed by some attribute declarations) to be used for the document element, or to a well-formed XML string.

Unless switch-to-new-documents option is turned off, this command also changes current node to the new document.

```
$scratch/> $t1 := create root
$t1> ls $t1
<?xml version="1.0" encoding="utf-8"?>
<root/>

$t1> $t2 := create "root id='r1'"
$t2> ls $t2
<?xml version="1.0" encoding="utf-8"?>
<root id="r1"/>

$t2> create "<root id='r0'>Just a <b>test</b></root>"
/> ls /
<?xml version="1.0" encoding="utf-8"?>
<root id='r0'>Just a <b>test</b></root>
```

See Also

open, clone

2.15. debug

Usage

```
debug
```

Description

Turn on debugging messages.

This is equivalent to setting `$DEBUG` variable to 1.

See Also

nodebug

2.16. declare-ns

Usage

```
declare-ns expression expression
```

Description

This command takes one or two arguments: prefix and URI, both evaluated as names. It creates a namespace declaration of the form `xmlns:prefix="URI"` on the current node. The command produces an error if the prefix is already declared in the scope of the current node with a different namespace URI.

See Also

set-ns, change-ns-uri, change-ns-prefix, namespaces

2.17. def

Usage

```
def subroutine [$variable ...] command-or-block
```

Aliases

```
define
```

Description

Define a new XSH2 sub-routine named subroutine. The subroutine may require zero or more parameters. These are declared as a whitespace-separated list of *parametric variables*. The body of the subroutine is specified as a command-or-block.

A sub-routine can be invoked directly by its name followed by its arguments just as any XSH2 command, or indirectly using the call command followed by an expression evaluating to the routine name and sub-routine arguments. Sub-routine arguments can be arbitrary expressions. These expressions are evaluated *prior* the sub-routine's code execution and are assigned to the sub-routine's parametric variables in the respective order. The number of parameter variables in a sub-routine definition and the number of arguments in a call to it must match. Calling a sub-routine with less or more arguments than declared is a run-time error.

Parametric variables are lexical variables within the sub-routine body as if they were declared with my.

Note that a subroutine has to be defined before it is first called (in terms of execution -- depending on the structure of the program, the actual definition of the sub-routine must not necessarily precede all references to it).

```
def l3 $nodes {
  ls --depth 3 $nodes; # list given nodes upto depth 3
}
l3 //chapter;           # direct call
$subref = 'l3';
call $subref //chapter; # in-direct call
```

Example 26. Commenting and un-commenting pieces of document

```
def comment
    $n      # nodes to move to comments
    $mark   # maybe some handy mark to recognize such comments
{
  foreach $n {
    if ( . = ../@* ) {
      echo "Warning: attribute nodes are not supported!";
    } else {
      echo "Commenting out:";
      ls --depth 0 .;
      add comment concat($mark,xsh:serialize(.)) replace .;
    }
  }
}

def uncomment $n $mark {
  foreach $n {
    if (. = ../comment()) { # is this node a comment node
      local $string = substring-after(.,"$mark");
      add chunk $string replace .;
    } else {
      echo "Warning: Ignoring non-comment node:";
      ls --depth 0 .;
    }
  }
}

# comment out all chapters with no paragraphs
comment //chapter[not(para)] "COMMENT-NOPARA";

# uncomment all comments stamped with COMMENT-NOPARA
$mark="COMMENT-NOPARA";
uncomment //comment()[starts-with(.,"$mark")] $mark;
```

See Also

call, return, my, local

2.18. defs

Usage

```
defs
```

Description

List names and parametric variables for all user-defined XSH2 subroutines.

See Also

def, variables

2.19. do

Usage

```
do command-or-block
```

Description

Execute command-or-block. This command is probably only useful when one wants to redirect output of more than one command.

See Also

command-or-block

2.20. doc-info

Usage

```
doc-info [document]
```

Aliases

```
doc_info
```

Description

In the present implementation, this command displays information provided in the `<?xml ...?>` declaration of a document: `version, encoding, standalone`, plus information about level of `gzip` compression of the original XML file and the original XML file URI.

See Also

set-enc, set-standalone

2.21. documents

Usage

```
documents
```

Aliases

```
files, docs
```

Description

Try to identify open documents and list their URIs and variables that contain them.

See Also

open, close

2.22. dtd

Usage

```
dtd [document]
```

Description

Print external or internal DTD for a given document. If used without arguments prints DTD of the current document.

See Also

set-dtd, validate

2.23. edit

Usage

```
edit [--editor|:E filename] [--all|:A] [--noindent|:n] [--recover|:r] [--
keep-blanks|:k] [--allow-empty|:0] [--no-coment|:q] [--encoding|:e encoding]
expression
```

Description

This command may be used to interactively edit parts of a XML document directly in your favorite editor.

A given expression is evaluated to a node-list and the first the first resulting node is opened in an external editor as a XML fragment. When the editor exits, the (possibly modified) fragment is parsed and returned to the original document. Unless `--no-comment` (`:q`) flag is used, the XML fragment is preceded with a XML comment specifying canonical XPath of the node being edited.

The command returns a node-list consisting of nodes that resulted from parsing the individual edits.

`--editor` or `:E` option may be used to specify external editor command. If not specified, environment variables `EDITOR` and `VISUAL` are tried first, then `vi` editor is used as a fallback.

If `--all` or `:A` flag is present, all nodes from the node-list are opened in the editor, one at a time.

If `--recover` or `:r` is specified, the parser tries to recover from possible syntax errors when parsing the resulting XML.

`--keep-blanks` or `:b` option may be used to force the parser to include ignorable white space.

If the result saved by the editor is empty, the interactive XSH2 shell asks user to confirm this was correct. This confirmation can be suppressed using `--allow-empty` or `:0` (zero) options.

The `--encoding` or `:e` parameter can be used to specify character encoding to use when communicating with the external editor.

Example 27. Edit all chapter elements (one by one) with emacs

```
edit --editor 'emacs -nw' --encoding iso-8859-1 --all //chapter
```

2.24. edit-string

Usage

```
edit-string [--editor|:E filename] [--encoding|:e encoding] [--allow-empty|:0]
expression
```

Description

Evaluating given expression to a string, save it in a temporary file, open the file an external editor as a plain text, and when the editor exits, read and return the result. The `--editor` (`:E`) parameter can be used to provide an editor command, whereas `--encoding` (`:e`) can be used to specify character encoding used for communication with the editor. If the result is empty, the interactive XSH2 shell asks user for confirmation before returning the result in order to prevent data loss due to some unexpected error. To suppress this feature, use `--allow-empty` or `:0` flag.

2.25. empty-tags

Usage

```
empty-tags expression
```

Aliases

```
empty_tags
```

Description

If the value of expression is 1 (non-zero), empty tags are serialized as a start-tag/end-tag pair (`<foo></foo>`). This option affects both ls and save and possibly other commands. Otherwise, they are compacted into a short-tag form (`<foo/>`). Default value is `0`.

This command is equivalent to setting the `$EMPTY_TAGS` variable.

2.26. enc

Usage

```
enc [document]
```

Description

Print the original document encoding string. If no document is given, the current document is used.

See Also

set-enc

2.27. encoding

Usage

```
encoding encoding
```

Description

Set the default character encoding used for standard (e.g. terminal) output. This doesn't influence the encoding used for saving XML documents.

This command is equivalent to setting the `$ENCODING` variable.

See Also

query-encoding

2.28. eval

Usage

```
eval expression
```

Description

NOTE: This command has very different behavior from XSH1 eval alias for perl.

This command first evaluates a given expression to obtain a string, then evaluates this string as XSH2 code in the current context, returning the return value of the last evaluated command. This command raises an exception if either expression evaluates to invalid XSH2 code or if evaluating the code raises an exception.

Example 28. Evaluate "in-line" XSH snippets within a XML document

```
foreach //inline-xsh eval .;
```

2.29. exec

Usage

```
exec expression [expression ...]
```

Aliases

```
system
```

Description

This command executes given expression(s) as a system command and returns the exit code.

Example 29. Count words in "hallo wold" string, then print name of your machine's operating system.

```
exec echo hallo world;              # prints hallo world
exec "echo hallo word" | wc;        # counts words in hallo world
exec uname;                         # prints operating system name
```

2.30. exit

Usage

```
exit [expression]
```

Aliases

```
quit
```

Description

Exit xsh, optionally with an exit-code resulting from evaluation of a given expression.

WARNING: No files are saved on exit.

2.31. fold

Usage

```
fold [--depth|:d expression] [expression]
```

Description

This feature is still EXPERIMENTAL and may change in the future! Fold command may be used to mark elements with a `xsh:fold` attribute from the `http://xsh.sourceforge.net/xsh/` namespace. When listing the DOM tree using `ls --fold xpath`, elements marked in this way are folded to a given depth (default is 0 = fold immediately).

The option `--depth` (`:d`) may be used to specify the depth at which subtrees of given elements are to be folded.

If called without arguments, the command applies to the current element, otherwise the expression is evaluated to the node-list and folding is applied to all elements within this node-list.

```
xsh> fold --depth 1 //chapter
xsh> ls --fold  //chapter[1]
<chapter id="intro" xsh:fold="1">
  <title>...</title>
  <para>...</para>
  <para>...</para>
</chapter>
```

See Also

unfold, ls

2.32. foreach

Usage

```
foreach expression command|command-or-block foreach [my|local] $variable in
expression command|command-or-block
```

Aliases

```
for
```

Description

Evaluate given expression to a node-list and for each resulting node execute given command or command-or-block. If used without a loop $variable, the loop temporarily sets current node to the node being processed. Otherwise, the processed node is assigned to the loop variable.

The expression may be xpath as well as a perl-code. In the latter case, if used without a loop variable, the loop automatically converts Perl objects to nodes. No conversion is performed when a loop variable is used.

Example 30. Move all employee sub-elements in a company element into the first staff subelement of the same company

```
xsh> foreach //company xmove employee into staff[1];
```

Example 31. List content of all XML files in current directory

```
xsh> foreach my $filename in { glob('*.xml') } {
  $f := open $filename;
  do_something $f;
}
```

2.33. get

Usage

```
get expression
```

Aliases

```
exp, expr
```

Description

Calculate a given expression and return the value.

See Also

count

2.34. hash

Usage

```
$hash := hash expression expression
```

Description

This command takes two arguments: an expression computing a key from a given node (1st argument) and a node-set (2nd argument). For each node in the node-set, the key value is computed and the node is stored under the given key in the resulting hash. For a given key, the value stored in the hash table is a node-list consisting of all nodes for which the 1st expression evaluated to an object string-wise equal to the key. It is therefore possible to index more than one node under the same key.

The XPath function `xsh:lookup(varname,key)` can be used to retrieve values from hashes in XPath expressions.

Example 32. Index books by author

```
my $books_by_author := hash concat(author/firstname," ",author/surname) //book;
```

Example 33. Lookup books by Jack London.

```
ls { $books_by_author->{'Jack London'} };
ls xsh:lookup('books_by_author','Jack London');
```

See Also

xsh:lookup

2.35. help

Usage

```
help command|argument-type|xsh:xpath-function
```

Description

Print help on a given command, argument type or XPath extension function (use `xsh:` as a prefix to XPath extensions function names, e.g `help xsh:id2`).

2.36. if

Usage

```
if expression command  if expression command-or-block [ elsif command-or-block ]* [ else command-or-block ]
```

Description

Executes command-or-block if a given expression expression evaluates to a non-emtpty node-list, true boolean-value, non-zero number or non-empty literal. If the first expression fails, then `elsif` conditions are tested (if any) and the command-or-block corresponding to the first one of them which is true is executed. If none of the conditions is satisfied, an optional `else` command-or-block is executed.

Example 34. Display node type

```
def node_type %n {
  foreach (%n) {
    if ( . = self::* ) { # XPath trick to check if . is an element
      echo 'element';
    } elsif ( . = ../@* ) { # XPath trick to check if . is an attribute
      echo 'attribute';
    } elsif ( . = ../processing-instruction() ) {
      echo 'pi';
    } elsif ( . = ../text() ) {
      echo 'text';
    } elsif ( . = ../comment() ) {
      echo 'comment'
    } else { # well, this should not happen, but anyway, ...
      echo 'unknown-type';
    }
  }
}
```

Example 35. Check a environment variable

```
if { defined($ENV{HOME}) } lcd { $ENV{HOME} }
```

2.37. ifinclude

Usage

```
ifinclude [--encoding|:e encoding] filename
```

Description

Unless the file filename has already been included using either include of ifinclude, load the file and execute it as a XSH2 script.

Use `--encoding` or `:e` parameter to specify character encoding used in the included file.

See Also

include

2.38. include

Usage

```
include [--encoding|:e encoding] filename
```

Aliases

.

Description

Load a file named filename and execute it as a XSH2 script.

Use `--encoding` or `:e` parameter to specify character encoding used in the included file.

See Also

ifinclude

2.39. indent

Usage

```
indent expression
```

Description

If the value of expression is 1, saved and listed XML will be formatted using some (hopefully) ignorable whitespace. If the value is 2 (or higher), XSH2 will act as in case of 1, plus it will add a leading and a trailing linebreak to each text node.

Note, that since the underlying C library (libxml2) uses a hard-coded indentation of 2 space characters per indentation level, the amount of whitespace used for indentation can not be altered at runtime.

This command is equivalent to setting the `$INDENT` variable.

2.40. index

Usage

```
index [document]
```

Description

This command makes `libxml2` library to remember document-order position of every element node in the document. Such indexation makes XPath queries considerably faster on large documents (with thousands of nodes). The command should only be used on documents which don't change; modifying an indexed document might possibly lead to non-conformant behavior of later XPath queries on the document.

2.41. insert

Usage

```
insert [--namespace|:n expression] node-type expression location xpath
```

Aliases

```
add
```

Description

Works just like xinsert, except that the new node is attached only the first node matched.

See Also

xinsert, move, xmove

2.42. iterate

Usage

```
iterate xpath command-or-block
```

Description

Iterate works very much like a foreach loop with the same xpath expression, except that it evaluates the command-or-block as soon as a new node matching a given xpath is found. As a limitation, an xpath expression used with `iterate` may consist of one XPath step only, i.e. it may not contain an XPath step separator `/`.

A possible benefit of using `iterate` instead of foreach is some efficiency when iterating over huge node-sets. Since `iterate` doesn't compute the resulting node-set in advance, it doesn't have to 1) allocate extra memory for it and 2) (more importantly) doesn't have to sort the node-list in the document order (which tends to be slow on large node-sets, unless index is used). On the other hand, `iterate` suffers from a considerable speed penalty since it isn't implemented in C (unlike libxml2's XPath engine).

Author's experience shows that, unless index is used, `iterate` beats foreach in speed on large node-lists (>=1500 nodes, but your milage may vary) while foreach wins on smaller node-lists.

The following two examples give equivalent results. However, the one using `iterate` may be faster if the number of nodes being counted is huge and document order isn't indexed.

Example 36. Count inhabitants of the kingdom of Rohan in productive age

```
cd rohan/inhabitants;

iterate child::*[@age>=18 and @age<60] { perl $productive++ };

echo "$productive inhabitants in productive age";
```

Example 37. Using XPath

```
$productive=count(rohan/inhabitants/*[@age>=18 and @age<60]);

echo "$productive inhabitants in productive age";
```

Hint: use e.g. `| time cut` pipe-line redirection to benchmark a XSH2 command on a UNIX system.

See Also

foreach, index, next, prev, last

2.43. keep-blanks

Usage

```
keep-blanks expression
```

Aliases

```
keep_blanks
```

Description

Allows you to turn on/off preserving the parser's default behavior of preserving all whitespace in the document. Setting this option to 0, instructs the XML parser to ignore whitespace occurring between adjacent element nodes, so that no extra text nodes are created for it.

This command is equivalent to setting the `$KEEP_BLANKS` variable.

2.44. last

Usage

```
last [expression]
```

Description

The last command is like the break statement in C (as used in loops); it immediately exits an enclosing loop. The optional expression argument may evaluate to a positive integer number that indicates which level of the nested loops to quit. If this argument is omitted, it defaults to 1, i.e. the innermost loop.

Using this command outside a subroutine causes an immediate run-time error.

See Also

foreach, while, iterate, next, last

2.45. lcd

Usage

```
lcd expression
```

Aliases

```
chdir
```

Description

Changes the filesystem working directory to expression, if possible. If expression is omitted, changes to the directory specified in HOME environment variable, if set; if not, changes to the directory specified by LOGDIR environment variable.

2.46. lineno

Usage

```
lineno [expression]
```

Description

`lineno` command prints line numbers of all nodes in a given node-list. Note however, that `libxml2` only stores line number information for element nodes.

See Also

locate

2.47. load-ext-dtd

Usage

```
load-ext-dtd expression
```

Aliases

```
load_ext_dtd
```

Description

Non-zero expression instructs the XML parser to load external DTD subsets declared in XML documents. This option is enabled by default.

This command is equivalent to setting the `$LOAD_EXT_DTD` variable.

2.48. local

Usage

```
local $variable = xpathlocal $variable [ $variable ... ]
```

Description

This command acts in a very similar way as assign does, except that the variable assignment is done temporarily and lasts only for the rest of its enclosing command-or-block or subroutine. At the end of the enclosing block or subroutine, the original value is restored. This also reverts any later usual assignments to the variable done occurring before the end of the block. This command may also be used without the assignment part.

Note, that the variable itself remains global in the sense that it is still visible to any subroutine called subsequently from the same block. Unlike my declaration, it does not *create* a new lexically scoped variable.

Hint for Perl programmers: `local` in XSH2 works exactly as `local` in Perl.

See Also

assign, my, def

2.49. locate

Usage

```
locate [--id|:i] xpath
```

Description

Print canonical XPaths leading to nodes matched by a given xpath.

If flag `--id` (`:i`) is used then ID-based shortcuts are allowed in the resulting location paths. That means that if the node or some of its ancestors has an ID attribute (either `xml:id` or one specified in a DTD) then the corresponding segment of the canonical location path is replaced by the `id()` function which jumps directly to an element based on its ID.

See Also

pwd

2.50. ls

Usage

```
ls [--fold|:f] [--fold-attrs|:A] [--indent|:i | --no-indent|:n] [--depth|:d
expression] [expression]
```

Aliases

```
list
```

Description

Print XML representation of a given expression, in particular, if used with an xpath, list parts of the document matching given expression.

If used without an argument, current node is listed to the depth 1 (see below).

`--depth` or `:d` argument may be used to specify depth of the XML listing. If negative, the listing depth is unlimited. All content below the specified depth is replaced with an ellipsis (`...`).

`--fold` or `:f` option makes the listing fold elements marked using the fold command are folded, i.e. listed only to the depth specified in the folding mark.

`--fold-attrs` or `:A` option avoids listing of attributes of the folded elements (i.e. elements on the lowest level of listing). Folded attributes are replaced with ellipsis (`...`).

`--indent` (`:i`) and `--no-indent` (`:n`) may be used to enforce/suppress indentation, overriding current setting (see command indent).

Unless in quiet mode, this command also prints the number of (top-level) nodes listed.

See Also

count, fold, unfold

2.51. map

Usage

```
map expression expression
```

Description

NOTE: THE SEMANTICS OF COMMAND HAS CHANGED IN 2.1.0

This command provides an easy way to transform node's data (content) using arbitrary expression. It takes two arguments: a mapping expression and a node-list.

First the second argument is evaluated to a node-list. For each of the nodes, the mapping expression is evaluated and the result is used to replace the original content of the node. The node is made the context node for the time of evaluation of the mapping expression. Moreover, if the expression is a Perl code, it gets the original text content in the variable `$_`.

Note that if the processed node is an element than the mapping expression may even produce nodes which are then copied into the element discarding any previous content of the element.

If the mapping expression returns an undefined value for a node, then its content is kept untouched.

`--in-place` (`:i`) flag: if the expression is a Perl code, then it is sometimes convenient to change the value in place. In that case use this flag to indicate that the result should to be taken from the `$_` variable rather than from the value of the expression itself. Without this flag, `$_` is read-only.

`--reverse` (`:r`) flag instruct the map to process the nodelist in reversed order.

Example 38. Capitalizes all hobbit names

```
map { ucfirst($_) } //hobbit/@name;
```

Example 39. Changes Goblins to Orcs in all hobbit tales (\b matches word boundary).

```
map :i { s/\bgoblin\b/orc/gi } //hobbit/tale/text();
```

Example 40. Recompute column sums in the last row of row-oriented table

```
map sum(/table/row[position()<last()]/
cell[count(xsh:current()/preceding-sibling::cell)+1])
/table/row[last()]/cell;
```

Example 41. The following commands do all about the same:

```
wrap --inner Z //*;
map --reverse xsh:parse(concat("<Z>",xsh:serialize(node()),"</Z>")) //*;
map xsh:parse(concat("<Z>",xsh:serialize(node()),"</Z>")) { reverse xpath('//*') };
```

Note that in the last example we use `:r` (or Perl `reverse` function) to reverse the node list order so that child nodes get processed before their parents. Otherwise, the child nodes would be replaced by parent's new content before the processing could reach them.

See Also

rename

2.52. move

Usage

```
move xpath location xpath
```

Aliases

`mv`

Description

`move` command acts exactly like copy, except that it *removes* the source nodes after a successful copy. Remember that the moved nodes are actually *different nodes* from the original ones (which may not be obvious when moving nodes within a single document into locations that do not require type conversion). So, after the move, the original nodes don't belong to any document and are automatically destroyed unless some variable still contains to them.

This command returns a node-list consisting of nodes it created on the target locations.

See copy for more details on how the copies of the moved nodes are created.

See Also

xmove, copy, xcopy, insert, xinsert

2.53. my

Usage

```
my $variable [$var2 ...];my $variable = expression;
```

Description

Same as in Perl: a "my" declares the listed variables to be local (lexically) to the enclosing block, or sub-routine.

See Also

local

2.54. namespaces

Usage

```
namespaces [--registered|:r] [expression]
```

Description

For each node in a given node-list lists all namespaces that are valid the scope of the node. Namespaces are listed in the form of `xmlns:prefix="uri"` declarations, preceded by a canonical xpath of the corresponding node on a separate line.

If `--registered` or `:r` flag is used, list also namespaces registered with the register-namespace command in XSH syntax.

If called without the `--registered` flag and no xpath is given, lists namespaces in the scope of the current node.

2.55. next

Usage

```
next [expression]
```

Description

`next` is like the continue statement in C; it starts the next iteration of an enclosing loop. The command may be used with an optional argument evaluating to a positive integer number indicating which level of the nested loops should be restarted. If the argument is omitted, it defaults to 1, i.e. the innermost loop.

Using this command outside a loop causes an immediate run-time error.

See Also

foreach, while, iterate, redo, last, prev

2.56. nobackups

Usage

```
nobackups
```

Description

Disable creating backup files on save.

This command is equivalent to setting the `$BACKUPS` variable to 0.

See Also

nobackups

2.57. nodebug

Usage

```
nodebug
```

Description

Turn off debugging messages.

This is equivalent to setting `$DEBUG` variable to 0.

See Also

debug

2.58. normalize

Usage

```
normalize expression
```

Description

`normalize` evaluates given expression to a node-list and puts all text nodes in the full depth of the sub-tree underneath each node in the node-list into a "normal" form where only structure (e.g., elements, comments, processing instructions, CDATA sections, and entity references) separates text nodes, i.e., there are neither adjacent text nodes nor empty text nodes.

Note, that most XSH2 commands automatically join adjacent text nodes.

2.59. open

Usage

```
$doc := open [--format|:F html|xml|docbook] [--file|:f | --pipe|:p | --
string|:s] [--switch-to|:w | --no-switch-to|:W] [--validate|:v | --no-valid-
ate|:V] [--recover|:r | --no-recover|:R] [--expand-entities|:e | --no-expand-
entities|:E] [--xinclude|:x | --no-xinclude|:X] [--keep-blanks|:b | --no-
keep-blanks|:B] [--pedantic|:n | --no-pedantic|:N] [--load-ext-dtd|:d | --
no-load-ext-dtd|:D] [--complete-attributes|:a | --no-complete-attributes|:A]
expression
```

Description

Parse a XML, HTML or SGML DOCBOOK document from a file or URL, command output or string and return a node-set consisting of the root of the resulting DOM tree.

`--format` (`:F`) option may be used to specify file format. Possible values are `xml` (default), `html`, and `docbook`. Note, however, that the support for parsing `DocBook` SGML files has been deprecated in recent `libxml2` versions.

`--file` (`:f`) instructs the parser to consider a given expression as a file name or URL.

`--pipe` (`:p`) instructs the parser to consider a given expression as a system command and parse its output.

`--string` (`:s`) instructs the parser to consider a given expression as a string of XML or HTML to parse.

`--switch-to` (`:w`) and `--no-switch-to` (`:W`) control whether the new document's root should become current node. These option override current global setting of switch-to-new-documents.

`--validate` (`:v`) and `--no-validate` (`:V`) turn on/off DTD-validation of the parsed document. These option override current global setting of validation.

`--recover` (`:r`) and `--no-recover` (`:R`) turn on/off parser's ability to recover from non-fatal errors. These option override current global setting of recovering.

`--expand-entities` (`:e`) and `--no-expand-entities` (`:E`) turn on/off entity expansion, overriding current global setting of parser-expands-entities.

`--xinclude` (`:x`) and `--no-xinclude` (`:X`) turn on/off XInclude processing, overriding current global settings of parser-expands-xinclude.

`--keep-blanks` (`:b`) and `--no-keep-blanks` (`:B`) control whether the parser should preserve so called ignorable whitespace. These option override current global setting of keep-blanks.

`--pedantic` (`:n`) and `--no-pedantic` (`:N`) turn on/off pedantic parser flag.

`--load-ext-dtd` (`:d`) and `--no-load-ext-dtd` (`:D`) control whether the external DTD subset should be loaded with the document. These option override current global setting of load-ext-dtd.

`--complete-attributes` (`:a`) and `--no-complete-attributes` (`:A`) turn on/off parse-time default attribute completion based on default values specified in the DTD. These option override current global setting of parser-completes-attributes.

```
$scratch/> $x := open mydoc.xml # open an XML document

# open a HTML document from the Internet
$h:=open --format html "http://www.google.com/?q=xsh"
# quote file name if it contains whitespace
$y := open "document with a long name with spaces.xml"

# use --format html or --format docbook to load these types
$z := open --format html index.htm

# use --pipe flag to read output of a command
$z := open --format html --pipe 'wget -O - xsh.sourceforge.net/index.html'

# use document variable to restrict XPath search to a
# given document
ls $z//chapter/title
```

2.60. parser-completes-attributes

Usage

```
parser-completes-attributes expression
```

Aliases

```
complete_attributes, complete-attributes, parser_completes_attributes
```

Description

If the expression is non-zero, the command makes XML parser add missing attributes with default values as specified in a DTD. By default, this option is enabled.

This command is equivalent to setting the `$PARSER_COMPLETES_ATTRIBUTES` variable.

2.61. parser-expands-entities

Usage

```
parser-expands-entities expression
```

Aliases

```
parser_expands_entities
```

Description

If the expression is non-zero enable the entity expansion during the parse process; disable it otherwise. If entity expansion is disabled, any external entity references in parsed documents are preserved as references. By default, entity expansion is enabled.

This command is equivalent to setting the `$PARSER_EXPANDS_ENTITIES` variable.

2.62. parser-expands-xinclude

Usage

```
parser-expands-xinclude expression
```

Aliases

```
parser_expands_xinclude
```

Description

If the expression is non-zero, the parser is allowed to expand XInclude tags immediately while parsing the document.

This command is equivalent to setting the `$PARSER_EXPANDS_XINCLUDE` variable.

See Also

process-xinclude

2.63. pedantic-parser

Usage

```
pedantic-parser expression
```

Aliases

```
pedantic_parser
```

Description

If you wish, you can make the XML parser little more pedantic by passing a non-zero expression to this command.

This command is equivalent to setting the `$PEDANTIC_PARSER` variable.

2.64. perl

Usage

```
perl perl-code
```

Description

Evaluate a given perl expression and return the result.

See Also

count

2.65. prev

Usage

```
prev [expression]
```

Description

This command is only allowed inside an `iterate` loop. It returns the iteration one step back, to the previous node on the iterated axis. The optional expression argument may be used to indicate to which level of nested loops the command applies to (default is 1).

See Also

iterate, redo, last, next

2.66. print

Usage

```
print [--nonl|:n] [--nospace|:s] [--stderr|:e] expression [expression ...]
```

Aliases

```
echo
```

Description

Evaluate given expression(s) and print the results (separated by a single space character). Expressions not containing any special characters, such as brackets, quotes, $, or @ are considered as bare words and evaluate to themselves.

`--nonl` or `:n` can be used to avoid printing a trailing new-line.

`--nospace` or `:s` suppresses printing additional spaces between individual arguments.

`--stderr` or `:e` causes the command to print on standard error output.

```
print foo   bar;  # prints "foo bar"
```

```
print "foo   bar";  # prints "foo   bar"
```

2.67. process-xinclude

Usage

```
process-xinclude [document]
```

Aliases

```
process_xinclude, process-xincludes, process_xincludes, xinclude, xincludes,
load_xincludes, load-xincludes, load_xinclude, load-xinclude
```

Description

Replace any xinclude tags in a given document with the corresponding content. See http://www.w3.org/TR/xinclude/ to find out more about XInclude technology.

See Also

parser-expands-xinclude

2.68. pwd

Usage

```
pwd [--id|:i]
```

Description

Print XPath leading to the current context node (equivalent to `locate .`).

If flag `--id` (`:i`) is used then ID-based shortcut is allowed in the resulting location path. That means that if the current node or some of its ancestors has an ID attribute (either `xml:id` or one specified in a DTD) then the corresponding segment of the canonical location path is replaced by the `id()` function which jumps directly to an element based on its ID.

See Also

locate, xsh:path

2.69. query-encoding

Usage

```
query-encoding encoding
```

Aliases

```
query_encoding
```

Description

Set the default query character encoding, i.e. encoding used when taking input from XSH2 prompt or standard input.

This command is equivalent to setting the `$QUERY_ENCODING` variable.

See Also

encoding

2.70. quiet

Usage

```
quiet
```

Description

Turn off verbose messages.

This command is equivalent to setting the `$QUIET` variable.

See Also

verbose

2.71. recovering

Usage

```
recovering expression
```

Description

If the expression evaluates to non-zero value, turn on the recovering parser mode on; turn it off otherwise. Defaults to off.

Note, that the in the recovering mode, validation is not performed by the parser even if the validation flag is on.

The recover mode helps to efficiently recover documents that are almost well-formed. This for example includes documents without a close tag for the document element (or any other element inside the document).

This command is equivalent to setting the `$RECOVERING` variable.

2.72. redo

Usage

```
redo [expression]
```

Description

`redo` restarts a loop block without evaluating the conditional again. The optional expression argument may evaluate to a positive integer number that indicates which level of the nested loops should be restarted. If omitted, it defaults to 1, i.e. the innermost loop.

Using this command outside a loop causes an immediate run-time error.

Example 42. Restart a higher level loop from an inner one

```
while ($i<100) {
  # ...
  foreach //para {
    # some code
    if $param {
      redo; # redo foreach loop
    } else {
      redo 2; # redo while loop
    }
  }
}
```

See Also

foreach, while, iterate, next, last

2.73. register-function

Usage

```
register-function expression perl-code
```

Aliases

```
function, regfunc
```

Description

EXPERIMENTAL!

Registers a given perl code as a new XPath extension function under a name provided in the first argument. XML::LibXML DOM API as well as all Perl functions pre-defined in the XML::XSH2::Map namespace may be used in the perl code for object processing. If the name contains a colon, then the first part before the colon must be a registered namespace prefix (see register-namespace) and the function is registered within the corresponding namespace.

2.74. register-namespace

Usage

```
register-namespace expression expression
```

Aliases

```
regns
```

Description

Registers the first argument as a prefix for the namespace given in the second argument. The prefix can later be used in XPath expressions.

2.75. register-xhtml-namespace

Usage

```
register-xhtml-namespace expression
```

Aliases

```
regns-xhtml
```

Description

Registers a prefix for the XHTML namespace `http://www.w3.org/1999/xhtml`. The prefix can later be used in XPath expressions.

2.76. register-xsh-namespace

Usage

```
register-xsh-namespace expression
```

Aliases

```
regns-xsh
```

Description

Registers a new prefix for the XSH2 namespace `http://xsh.sourceforge.net/xsh/`. The prefix can later be used in XPath expressions. Note, that XSH2 namespace is by default registered with the prefix `xsh`. This command is thus, in general, useful only when some document uses `xsh` prefix for a different namespace or if you want a shorter prefix.

2.77. remove

Usage

```
remove expression
```

Aliases

```
rm, prune, delete, del
```

Description

Unlink all nodes in a given node-list from their respective documents. Nodes, which are neither attached to a document or stored in a variable are automatically garbage-collected.

Returns a number of nodes removed.

Example 43. Get rid of all evil creatures.

```
xsh> del //creature[@manner='evil']
```

2.78. rename

Usage

```
rename nodename expression
```

Description

NOTE: THE SEMANTICS OF COMMAND HAS CHANGED IN 2.1.0

This command is very similar to the map command, except that it operates on nodes' names rather than their content. It changes name of every element, attribute or processing-instruction contained in the node-list specified in the second argument expression, according to the value of the nodename expression, which is evaluated in the context of each node in turn.

If the nodename is a Perl expression, then the name of the node is also stored into Perl's `$_` variable prior to evaluation.

The flag `--in-place` (`:i`) can be used to indicate that the new name should be collected from the `$_` variable rather than from the result of the expression itself.

The `--namespace` (`:n`) argument may be used to provide namespace for the renamed nodes.

`--reverse` (`:r`) flag instruct the map to process the nodelist in reversed order.

Note: if the expression nodename returns an undefined value for a particular node, the node's original name and namespace are preserved.

Example 44. Renames all Hobbits to Halflings

```
xsh> rename halfling //hobbit
```

Example 45. Make all elements and attributes uppercase (yack!)

```
xsh> rename { uc } (//*|//@*)
```

Example 46. Substitute dashes with underscores in all node names

```
xsh> rename :i { s/-/_/g } (//*|//@*)
```

Example 47. Make all elements start with the name of their parents

```
xsh> rename concat(local-name(parent::*),'.',local-name(.)) //*[parent::*]
```

See Also

map

2.79. return

Usage

```
return [expression]
```

Description

This command immediately stops the execution of a procedure it occurs in and returns the execution to the place of the script from which the subroutine was called. Optional argument may be used as a return value for the subroutine call.

Using this command outside a subroutine causes an immediate run-time error.

See Also

def, call

2.80. run-mode

Usage

```
run-mode
```

Aliases

```
run_mode
```

Description

Switch from the test-mode back to the normal execution mode.

This is equivalent to setting `$TEST_MODE` variable to 0.

See Also

test-mode

2.81. save

Usage

```
save [--format|:F html|xml] [--xinclude|:x] [--file|:f filename | --pipe|:p
filename | --string|:s | --print|:r ] [--subtree|:S] [--indent|:i | --no-in-
dent|:I] [--skip-dtd|:d | --no-skip-dtd|:D] [--skip-empty-tags|:t | --no-
skip-empty-tags|:T] [--skip-xmldecl|:x] [--encoding|:e encoding] document
```

Description

This takes a given document, serializes it to XML or HTML and either saves the result to its original file or another file (default), pipes it to an external command, prints it on standard output, or simply returns it. Without arguments it simply saves current document to its original file.

`--file|:f` option may be used to specify an output file-name. By default, the original document's file-name is used.

`--pipe|:p` option specifies, that the output should be piped to an external command specified as the option's argument.

`--print|:r` option specifies, that the output should be printed on standard output.

`--string|:s` option specifies, that the output should be returned by the command as a string. In this case, the result is always in UTF8, regardless on which encoding is specified in the document or using `--encoding` option.

The above four options are mutually exclusive.

`--format` option may be used to specify the output format. It's argument should be either `xml`, `html` or an expression evaluating to one of these. If not specified, XML output is assumed. Note, that a document should be saved as HTML only if it actually is a HTML document, otherwise the result would be an invalid XML instance. Note also, that the optional encoding parameter only forces character conversion; it is up to the user to declare the document encoding in the appropriate HTML <META> tag, if needed.

`--xinclude` automatically implies XML format and can be used to force XSH2 to save all already expanded XInclude sections back to their original files while replacing them with <xi:include> tags in the main XML file. Moreover, all material included within <include> elements from the http://www.w3.org/2001/XInclude namespace is saved to separate files too according to the `href` attribute, leaving only empty <include> element in the root file. This feature may be used to split the document to a new set of XInclude fragments.

If the `--subtree` (`:S`) flag is used, only the subtree of the specified node is saved instead of the complete document (this flag cannot be used with `--html` format).

`--indent` (`:i`) and `--no-indent` (`:n`) may be used to enforce/suppress indentation, overriding current global setting of indent.

`--skip-dtd` (`:d`) and `--no-skip-dtd` (`:D`) may be used to enforce/suppress skipping DTD declaration and internal subset on output, overriding current global setting of skip-dtd.

`--empty-tags` (`:t`) and `--no-empty-tags` (`:T`) may be used to override current global setting of empty-tags. `--no-empty-tags` instructs XSH2 to serialize elements with no child nodes as start-tag/end-tag pair `<element></element>` instead of using a short empty-tag form `<element/>`.

`--skip-xmldecl` (`:x`) instructs XSH2 to omit the XML declaration from the saved document. Note however, that XML declaration is obligatory for XML documents. It usually looks like `<?xml version="1.0" ...?>`.

`--backup` (`:b`) and `--no-backup` (`:B`) can be used to enforce/suppress creation of a backup file if target file already exists. Using these options overrides current global setting of backups.

`--encoding` followed by a encoding instructs XSH2 to save the document in the specified encoding. In case of XML output, the <?xml?> declaration is automatically changed accordingly.

Example 48. Use save to preview current HTML document in Lynx

```
save --format html --pipe 'lynx -stdin'
```

See Also

open, close, enc, documents

2.82. set

Usage

```
set xpath [xpath]
```

Description

This command provides very easy way to create or modify content of a document. It takes two XPath expressions. The first one should be a node location path which specifies the target node, the second is optional and provides new content for the target node. If a node matches the first XPath expression, then its content is replaced with the

given value. If no node matches, then XSH2 tries to magically extend the current document by adding nodes in order to add missing steps of the location path so as to make the expression match a node. This node is then populated with a copy of the content value (either text or, if the content xpath results in a node-list and the target node is an element, nodes).

Example 49. Try the following on an empty scratch document

```
$scratch/> ls /
<scratch/>
$scratch/> set scratch/@say "hallo world"
<scratch say="hello world"/>

$scratch/> set scratch/foo[2]/../foo[1]/following-sibling::bar/baz[3] "HALLO"
$scratch/> ls /
<?xml version="1.0" encoding="utf-8"?>
<scratch>
  <foo/>
    <bar>
      <baz/>
      <baz/>
      <baz>HALLO</baz>
    </bar>
  <foo/>
<scratch/>
```

Only a limited subset of XPath is currently supported by this command. Namely, the XPath expression must be a location path consisting of a /-separated sequence of one or more location steps along the child, sibling, or attribute axes. The node-test part of the expression cannot be neither a wildcard (`*`, `@*`, `prefix:*`, ...), nor the `node()` function. If a namespace prefix is used, then either the namespace must already be declared in the document or registered with XSH. Location steps may contain arbitrary predicates (filters), however, only a limited subset is supported for magically created nodes.

In particular, if a filter predicate of a location step specifies a position of a node (e.g. with `[4]`, or `[position()>3]`, etc), then the parser tries to automatically create empty siblings nodes until it finally creates one with for which the predicate is true.

Note, that this command only processes one location step at a time and always picks the first matching node. So, expressions like `/root/a/b` are treated as `/root/a[1]/b[1]`. This means that an expression `/root/a/b` will magically create element `<b>` in a first matching `<a>` even if some following `<a>` already contains a `<b>`.

To prevent this, either explicitly state that `b` must exist with e.g. `/root/a[b]/b` or make the corresponding element `<a>` the context node and use a relative location path:

```
for /root/a/b set b 'foo'
```

2.83. set-dtd

Usage

```
set-dtd [--internal] [--name|:n expression] [--public|:p expression] [--system|:s expression] [document]
```

Aliases

```
set_dtd
```

Description

Set external (default) or internal DTD for a given document. If no document is given, the current document is used. At least one of `--public` and `--system` options should be used to specify the PUBLIC and SYSTEM identifiers of the DTD. If `--name` parameter is not provided, name of the document root element is used as DTD name.

See Also

dtd, validate

2.84. set-enc

Usage

```
set-enc encoding [document]
```

Description

Changes character encoding of a given document. If no document is given, the command applies to the current document. This has two effects: changing the XMLDecl encoding declaration in the document prolog to display the new encoding and making all future save operations on the document default to the given charset.

```
xsh> ls
<?xml version="1.0" encoding="iso-8859-1"?>
<foo>...</foo>
xsh> set-enc "utf-8"
xsh> ls
<?xml version="1.0" encoding="utf-8"?>
<foo>...</foo>
xsh> save# saves the file in UTF-8 encoding
```

See Also

enc, doc-info

2.85. set-ns

Usage

```
set-ns [:p|--prefix expression] expression
```

Description

This command takes one argument, the namespace URI, possibly accompanied by a prefix provided in the option `--prefix :p`; both these expressions are evaluated as names. The command changes the namespace of the current element to a given namespace URI. The current node must be in the scope of a namespace declaration associating the namespace URI with a prefix; if prefix option is given, then one of such declarations must associate the particular given prefix with the namespace URI. If this condition is not met or the current node is neither element nor attribute, an error is issued. The command also changes the prefix of the current element accordingly.

See Also

declare-ns, change-ns-uri, change-ns-prefix, namespaces

2.86. set-standalone

Usage

```
set-standalone expression [document]
```

Description

Changes the value of `standalone` declaration in the XMLDecl prolog of a document. The expression should evaluate to either 1 or 0 or `'yes'` or `'no'`. The result of applying the command on other values is not specified. If no document is given, the command applies to the current document.

See Also

doc-info

2.87. set_filename

Usage

```
set_filename expression [document]
```

Description

Changes filename or URL associated with a given document (or the current document, if only one argument is specified). Document filename is initialized by the open command and used e.g. by save. It can be queried in XPath expressions using the xsh:filename function.

See Also

open, save, xsh:filename

2.88. settings

Usage

```
settings
```

Description

List current values of all XSH2 settings (such as validation flag or query-encoding).

`--variables` or `:v` flag enforces syntax which makes use of variable assignments. Otherwise, settings are listed in the form of XSH commands.

Example 50. Store current settings in your .xshrc

```
xsh> settings | cat > ~/.xshrc
```

2.89. skip-dtd

Usage

```
skip-dtd expression
```

Aliases

`skip_dtd`

Description

If the value of expression is 1 (non-zero), DTD DOCTYPE declaration is omitted from any further serializations of XML documents (including ls and save). Default value is `0`.

This command is equivalent to setting the `$SKIP_DTD` variable.

2.90. sort

Usage

```
$result := sort [ --key|:k expression ] --compare|:c perl-code expression$res-
ult := sort [ --key|:k expression ] [ --numeric|:n ] [ --descending|:d ] [
--locale|:l] expression
```

Description

This command sorts a given node-list, returning a node-list ordered according to a given key and ordering function.

`--key|:k` followed by an expression specifies the key to be computed for each member of the node-list and the result used as the sorting key. If omitted, keys are created by converting the nodes to string as if XPath expression `string(.)` was used.

`--numeric|:n` specifies, that keys should be compared by their numerical values (the default is string comparison).

`--descending|:d` specifies, that the result should be ordered in descending order (default is ascending).

`--locale|:l` forces using current locale settings for string comparison (default is no locale).

`--compare` argument followed by a perl-code allows to define a custom comparison method in a similar way to Perl `sort` command. The keys to be compared are passed to the code in variables `$a` and `$b`. The code is supposed to return 1 if the key in `$a` is greater than `$b`, 0 if the keys are equal and `-1` if `$a` is less than `$b`, depending on how the corresponding elements are to be ordered. It is a run-time error to use `--compare` together with either `--numeric` or `--descending`.

Example 51. Case-insensitive sort of a given node-list

```
$ordered := sort --key xsh:lc(.) $unordered;
```

Example 52. Reorder creature elements by name attribute in ascending order using Czech locale settings

```
perl {
# setup locale collating function
# Note, that the collating function must be UTF8 aware.
use POSIX qw(locale_h);
setlocale(LC_COLLATE,'cs_CZ.UTF-8');
};

xmove &{ sort :k@name :l * } into /middle-earth[1]/creatures;
```

Example 53. Sort a node-list by a pre-computed score (Perl-based sort)

```
$results := sort --numeric --descending --key { $scores{literal('@name')} } $players;
```

2.91. stream

Usage

```
stream [ --input-file|:f filename | --input-pipe|:p filename | --input-
string|:s expression] [ --output-file|:F filename | --output-pipe|:P filename
| --output-string|:S $variable ] select xpath command-or-block [ select xpath
command-or-block ... ]
```

Description

EXPERIMENTAL! This command provides a memory efficient (though slower) way to process selected parts of an XML document with XSH2. A streaming XML parser (SAX parser) is used to parse the input. The parser has two states which will be referred to as A and B below. The initial state of the parser is A.

In the state A, only a limited vertical portion of the DOM tree is built. All XML data coming from the input stream other than start-tags are immediately copied to the output stream. If a new start-tag of an element arrives, a new node is created in the tree. All siblings of the newly created node are removed. Thus, in the state A, there is exactly one node on every level of the tree. After a node is added to the tree, all the xpath expressions following the `select` keyword are checked. If none matches, the parser remains in state A and copies the start-tag to the output stream. Otherwise, the first expression that matches is remembered and the parser changes its state to B.

In state B the parser builds a complete DOM subtree of the element that was last added to the tree before the parser changed its state from A to B. No data are sent to the output at this stage. When the subtree is complete (i.e. the corresponding end-tag for its topmost element is encountered), the command-or-block of instructions following the xpath expression that matched is invoked with the root element of the subtree as the current context node. The commands in command-or-block are allowed to transform the whole element subtree or even to replace it with a different DOM subtree or subtrees. They must, however, leave intact all ancestor nodes of the subtree. Failing to do so can result in an error or unpredictable results.

After the subtree processing command-or-block returns, all subtrees that now appear in the DOM tree in the place of the original subtree are serialized to the output stream. After that, they are deleted and the parser returns to state A.

Note that this type of processing highly limits the amount of information the selecting XPath expressions can use. The first notable fact is, that elements can not be selected by their content. The only information present in the tree at the time of the XPath evaluation is the element's name and attributes plus the same information for all its ancestors (there is no information at all about its possible child nodes nor of the node's position within the list of its siblings).

The input parameters below are mutually exclusive. If non is given, standard input is processed.

`--input-file` or `:f` instructs the processor to stream from a given file.

`--input-pipe` or `:p` instructs the processor to stream the output of a given a command.

`--input-string` or `:s` instructs the processor to use the result of a given expression as the input to be processed.

The output parameters below are mutually exclusive. If none is given, standard output is used.

`--output-file` or `:F` instructs the processor to save the output to a given file.

`--output-pipe` or `:P` instructs the processor to pipe the output to a given command.

`--output-string` or `:S` followed by a variable name instructs the processor to store the result in the given variable.

2.92. strip-whitespace

Usage

```
strip-whitespace expression
```

Aliases

```
strip_whitespace
```

Description

`strip-whitespace` removes all leading and trailing whitespace from given nodes. If applied to an element node, it removes all leading and trailing child whitespace-only text nodes and CDATA sections.

2.93. switch-to-new-documents

Usage

```
switch-to-new-documents expression
```

Aliases

```
switch_to_new_documents
```

Description

If non-zero, XSH2 changes current node to the document node of a newly open/created files every time a new document is opened or created with open or create. Default value for this option is 1.

This command is equivalent to setting the `$SWITCH_TO_NEW_DOCUMENTS` variable.

2.94. test-mode

Usage

```
test-mode
```

Aliases

```
test_mode
```

Description

Switch into a mode in which no commands are actually executed and only command syntax is checked.

This is equivalent to setting `$TEST_MODE` variable to 1.

See Also

run-mode

2.95. throw

Usage

```
throw expression
```

Description

This command throws and exception containing error message given by the obligatory expression argument. If the exception is not handled by some surrounding try block, the execution is stopped immediately and the error message is printed.

See Also

try

2.96. try

Usage

```
try command-or-block catch [[local|my] $variable] command-or-block
```

Description

Execute the command-or-block following the `try` keyword. If an error or exception occurs during the evaluation, execute the `catch` command-or-block. If the `catch` keyword is followed by a variable (possibly localized for the following block using my or local) and the `try` block fails with an exception, the error message of the exception is stored to the variable before the `catch` block is executed.

The throw command as well as an equivalent Perl construction `perl { die "error message" }` allow user to throw custom exceptions.

Unless exception is raised or error occurs, this command returns the return value of the `try` block; otherwise it returns the return value of the `catch` block.

Example 54. Handle parse errors

```
try {
  $doc:=open --format xml $input;
} catch {
  try {
    echo "XML parser failed, trying HTML";
    $doc := open --format html $input;
  } catch my $error {
    echo "Stopping due to errors: $error";
    exit 1;
  }
}
```

See Also

throw

2.97. undef

Usage

```
undef [subroutine | $variable]
```

Aliases

```
undefine
```

Description

This command can be used to undefine previously defined XSH2 subroutines and variables.

See Also

close, def

2.98. unfold

Usage

```
unfold expression
```

Description

This feature is still EXPERIMENTAL!

Unfold command removes `xsh:fold` attributes from all given elements. Such attributes are usually created by previous usage of fold. Be aware, that `xmlns:xsh` namespace declaration may still be present in the document even when all elements are unfolded.

See Also

fold, ls

2.99. unless

Usage

```
unless expression command  unless expression command-or-block [ else command-or-block ]
```

Description

Like if but negating the result of the expression. Also, unlike if, `unless` has no `elsif` block.

See Also

if

2.100. unregister-function

Usage

```
unregister-function expression
```

Aliases

```
unregfunc
```

Description

EXPERIMENTAL! Unregister XPath extension function of a given name previously registered using register-function.

2.101. unregister-namespace

Usage

```
unregister-namespace expression
```

Aliases

```
unregns
```

Description

Unregisters given namespace prefix previously registered using register-namespace. The prefix can no longer be used in XPath expressions unless declared within the current scope of the queried document.

2.102. validate

Usage

```
validate [--yesno|:q] [document]validate [--yesno|:q] [--dtd|:D | --relaxng|:R
| --schema|:S] --file|:f filename] [document]validate [--yesno|:q] [--dtd|:D
| --relaxng|:R | --schema|:S] --string|:s expression] [document]validate [-
-yesno|:q] [--dtd|:D | --relaxng|:R | --schema|:S] --doc|:d document] [docu-
ment]validate [--yesno|:q] [--dtd|:d] --public|:p expression [--file|:f ex-
pression] [document]
```

Description

This command validates the current document or a document specified in the argument against a DTD, RelaxNG or XSD schema. If `--yesno` or `:q` is specified, only prints `yes` and returns 1 if the document validates or `no` and returns 0 if it does not. Without `--yesno`, it throws an exception with a complete validation error message if the document doesn't validate.

`--dtd` or `:D` forces DTD validation (default).

`--relaxng` or `:R` forces RelaxNG validation. Only XML RelaxNG grammars are supported.

`--schema` or `:S` forces W3C XML Schema validation (XSD). Support for schema validation may still be incomplete (see libxml2 home page [http://xmlsoft.org] for more details).

A DTD subset can be specified by its PUBLIC identifier (with `--public`), by its SYSTEM identifier (with `--file`), or as a string (with `--string`). If none of these options is used, validation is performed against the internal or external DTD subset of the document being validated.

RelaxNG grammars and XML Schemas can either be specified either as a filename or url (with `--file`), as a string containing (with `--string`), or as a document currently open in XSH2 (with `--doc`).

```
$mydoc := open "test.xml"

# in all examples below, mydoc can be omitted

validate --yesno $mydoc; # validate against the document's DOCTYPE

validate --public "-//OASIS//DTD DocBook XML V4.1.2//EN" $mydoc

validate --file "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" $mydoc

validate --relaxng --file "test.rng" $mydoc

validate --relaxng --string $relaxschema $mydoc

$rng := open "test.rng"

validate --relaxng --doc $rng $mydoc

validate --schema --file "test.xsd" $mydoc

validate --schema --string $xsdschema $mydoc

$xsd := open "test.xsd"

validate --schema --doc $xsd $mydoc
```

See Also

dtd

2.103. validation

Usage

```
validation expression
```

Description

Turn on validation during the parse process if the expression is non-zero or off otherwise. Defaults to off.

This command is equivalent to setting the `$VALIDATION` variable.

2.104. variables

Usage

```
variables
```

Aliases

```
vars, var
```

Description

List all global variables and their current values.

See Also

documents, defs

2.105. verbose

Usage

```
verbose
```

Description

Turn on verbose messages (default).

This is equivalent to setting `$QUIET` variable to 0.

See Also

quiet

2.106. version

Usage

```
version
```

Description

Prints program version plus version numbers of the most important libraries used.

2.107. while

Usage

```
while expression command-or-block
```

Description

Execute the command-or-block as long as the given expression evaluates to a non-emtpty node-list, true boolean-value, non-zero number or non-empty literal.

Example 55. The commands in this example do the same thing

```
xsh> while /table/row remove /table/row[1];
xsh> remove /table/row;
```

2.108. wrap

Usage

```
wrap [--namespace expression] [ [--inner] | [--while|:W expression] [--until|:U expression] [--skip-whitespace|:w] [--skip-comments|:c] [--skip-pi|:p] ] expression xpath
```

Description

For each node matching the xpath argument, this command creates a new element node according to a given expression (in the same way as xinsert does) which replaces the matching node, and moves the matching node into this newly created element. If namespace expression is given, the namespace is applied on the created element. The command returns a node-list consisting of the elements created.

With `--inner` (or `:i`) flag the command wraps children nodes of the matching node rather than the node it self the following sense: for each matching node a new element node is created, but this time it is placed into the matching node and all previous children of the matching node are moved into the newly created node. In this mode, all non-element matching nodes are ignored. This flag cannot be used together with `--while` and `--until`, which we describe next.

`--while` (`:W`) and/or `--until` (`:U`) arguments can be provided in order to move a sequence of adjacent siblings following the matching node into the newly created element. In this way the newly created element wraps not just the matching node itself but a range of nodes starting at the matching node and ending either before a first following node matching the expression of `--until`, or before a first following node not matching the expression of `--while`, or at the last sibling if neither of the prior apply. Both these expressions are evaluated in the context of the currently tested sibling and prior to the creation of the wrapping element. The context position for these expressions is 1 at the first sibling following the matching node and increases with each tested sibling; the context size is the number of all siblings following the matching node. It is important to mention that siblings wrapped in this way are excluded from further processing by wrap even if included in the node-list produced by the xpath argument. This allows to easily wrap certain adjacent elements without worrying about some elements being wrapped multiple times (for example, `wrap :W x y //x` wraps each sequence of adjacent elements <x> in a <y>).

`--skip-whitespace` (`:w`), `--skip-comments` (`:c`), and `--skip-pi` (`:p`) can be used in combination with `--while` (`:W`) and/or `--until` (`:U`) to skip testing the expressions on white-space text nodes, comments, and/or processing instruction, respectively. Such nodes are only included in the wrapped range if followed by a sibling that is to be wrapped.

```
$scratch/> ls /;
<?xml version="1.0" encoding="utf-8"?>
<scratch/>

$scratch/> wrap 'foo' *;
$scratch/> insert attribute 'bar=baz' into /foo;
$scratch/> insert text 'some text' into //scratch;
$scratch/> wrap --namespace 'http://foo/bar' 'a:A' //@*;
$scratch/> $wrapper := wrap 'text aaa="bbb"' //text();
$scratch/> wrap '<elem ccc=ddd>' //*;
$scratch/> ls /;
<?xml version="1.0" encoding="utf-8"?>
<elem ccc="ddd">
  <foo xmlns:a="http://foo/bar">
    <elem ccc="ddd">
      <scratch>
        <elem ccc="ddd">
          <text aaa="bbb">some text</text>
        </elem>
      </scratch>
    </elem>
```

```
    <elem ccc="ddd">
      <a:A xmlns:a="http://foo/bar" bar="baz"/>
    </elem>
  </foo>
</elem>

$scratch/> ls $wrapper;
<text aaa="bbb">some text</text>


$scratch/> wrap --inner bar //foo
$scratch/> ls /;
<?xml version="1.0" encoding="utf-8"?>
<elem ccc="ddd">
  <foo xmlns:a="http://foo/bar">
    <bar>
      <elem ccc="ddd">
        <scratch>
          <elem ccc="ddd">
            <text aaa="bbb">some text</text>
          </elem>
        </scratch>
      </elem>
      <elem ccc="ddd">
        <a:A xmlns:a="http://foo/bar" bar="baz"/>
      </elem>
    </bar>
  </foo>
</elem>
```

Example 56. Wrapping a range of adjacent nodes

```
# prepare the test document
$scratch/> rm /scratch/node(); # cleanup the document
$scratch/> set /scratch/li[5]; # create 5 <li> elements
$scratch/> set /scratch/li[3]/following-sibling::li; # add <br/> after the 3rd <li>
$scratch/> for //li set . position(); # number the <li> elements
$scratch/> ls /
<?xml version="1.0" encoding="utf-8"?>
<scratch>
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <br/>
  <li>4</li>
  <li>5</li>
</scratch>
# wrap adjacent elements <li> into an <ol>
$scratch/> wrap --skip-whitespace --while self::li ol //li;
$scratch/> ls /
<?xml version="1.0" encoding="utf-8"?>
<scratch>
  <ol>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ol>
  <br/>
  <ol>
    <li>4</li>
    <li>5</li>
  </ol>
</scratch>
```

See Also

xinsert, insert, move, xmove

2.109. wrap-span

Usage

```
wrap-span [--namespace expression] expression expression expression
```

Aliases

```
wrap_span
```

Description

This command is very similar to wrap command, except that it works on spans of nodes. It wraps spans (i.e. sequence of adjacent nodes between (and including) a start node and an end node) with a new element whose name is specified as the first argument. Nodes within each span must have the same parent node. The spans to be wrapped are defined by a pair of node-lists in the second and third argument. The first node-list specifies the start node of one or more spans, while the second node-list should contain the corresponding end nodes. The two node-lists must evaluate to the exactly same number of nodes, otherwise a runtime error is reported. The N'th span is then defined as a span starting on the N'th node in the start node-list and ending at the N'th node in the end node-list.

All nodes within the spans are removed from the document and placed into the newly generated elements. The wrapping elements are put back into the document tree at the positions previously occupied by the node-spans.

The command returns a node-list containing the newly created wrapping elements.

```
xsh $scratch/> $foo := create { "<root>\n<a/><b/>\n<a/><b/>\n<a/><b/>\n</root>" };
xsh $foo/> wrap-span 'span' //a //b;
xsh $foo/> ls /;
<?xml version="1.0" encoding="utf-8"?>
<root>
<span><a/><b/></span>
<span><a/><b/></span>
<span><a/><b/></span>
</root>
```

See Also

xinsert, insert, move, xmove

2.110. xcopy

Usage

```
xcopy [--respective|:r] [--preserve-order|:p] expression location expression
```

Aliases

```
xcp
```

Description

xcopy is similar to copy, but copies *all* nodes in the first node-list expression to *all* destinations determined by the location directive relative to the second node-list expression. See copy for detailed description of `xcopy` arguments.

If `--respective|:r` option is used, then the target node-list expression is evaluated in the context of the source node being copied.

The `--preserve-order|:p` option can be used to ensure that the copied nodes are in the same relative order as the corresponding source nodes. Otherwise, if location is `after` or `prepend`, the relative order of the copied nodes will be reversed, because source nodes are placed to the target location one by one.

Example 57. Copy all middle-earth creatures within the document $a into every world of the document $b.

```
xsh> xcopy $a/middle-earth/creature into $b//world
```

See Also

copy, move, xmove, insert, xinsert

2.111. xinsert

Usage

```
xinsert [--namespace expression] node-type expression location xpath
```

Aliases

`xadd`

Description

Create new nodes of the node-type given in the 1st argument of name specified in the 2nd argument and insert them to locations relative to nodes in the node-list specified in the 4th argument.

For element nodes, the the 2nd argument expression should evaluate to something like "<element-name att-name='attvalue' ...>". The < and > characters are optional. If no attributes are used, the expression may simply consist the element name. Note, that in the first case, the quotes are required since the expression contains spaces.

Attribute nodes use the following syntax: "att-name='attvalue' [...]".

For the other types of nodes (text, cdata, comments) the expression should contain the node's literal content. Again, it is necessary to quote all whitespace and special characters as in any expression argument.

The location argument should be one of: `after`, `before`, `into`, `replace`, `append` or `prepend`. See documentation of the location argument type for more detail.

Optionally, for element and attribute nodes, a namespace may be specified with `--namespace` or `:n`. If used, the expression should evaluate to the desired namespace URI and the name of the element or attribute being inserted *must have a prefix*.

The command returns a node-list consisting of nodes it created.

Note, that instead of `xinsert`, you can alternatively use one of xsh:new-attribute, xsh:new-cdata, xsh:new-chunk, xsh:new-comment, xsh:new-element, xsh:new-element-ns, xsh:new-pi, and xsh:new-text together with the command xcopy.

Example 58. Give each chapter a provisional title element.

```
xsh> my $new_titles := xinsert element "<title font-size=large underline=yes>" \
  into /book/chapter
xsh> xinsert text "Change me!" into $new_titles;
```

Example 59. Same as above, using xcopy and xsh:new-... instead of xinsert

```
xsh> my $new_titles := xcopy
xsh:new-element("title","font-size","large","underline","yes") \
  into /book/chapter
xsh> xcopy xsh:new-text("Change me!") into $new_titles;
```

See Also

insert, move, xmove

2.112. xmove

Usage

```
xmove [--respective|:r] [--preserve-order|:p] xpath location xpath
```

Aliases

xmv

Description

Like xcopy, except that `xmove` *removes* the source nodes after a successful copy. Remember that the moved nodes are actually *different nodes* from the original ones (which may not be obvious when moving nodes within a single document into locations that do not require type conversion). So, after the move, the original nodes don't belong to any document and are automatically destroyed unless still contained in some variable.

This command returns a node-list consisting of all nodes it created on the target locations.

If `--respective|:r` option is used, then the target node-list expression is evaluated in the context of the source node being copied.

The `--preserve-order|:p` option can be used to ensure that the copied nodes are in the same relative order as the corresponding source nodes. Otherwise, if location is `after` or `prepend`, the relative order of the copied nodes will be reversed, because source nodes are placed to the target location one by one.

See xcopy for more details on how the copies of the moved nodes are created.

The following example demonstrates how `xmove` can be used to get rid of HTML `<font>` elements while preserving their content. As an exercise, try to figure out why simple `foreach //font { xmove node() replace . }` would not work here.

Example 60. Get rid of all <font> tags

```
while //font {
  foreach //font {
    xmove node() replace .;
  }
}
```

See Also

move, copy, xcopy, insert, xinsert

2.113. xpath-axis-completion

Usage

```
xpath-axis-completion expression
```

Aliases

```
xpath_axis_completion
```

Description

The following values are allowed: `always`, `never`, `when-empty`. Note, that all other values (including 1) work as `never`!

If the expression evaluates to `always`, TAB completion for XPath expressions always includes axis names.

If the expression evaluates to `when-empty`, the TAB completion list for XPath expressions includes axis names only if no element name matches the completion.

If the expression evaluates to `never`, the TAB completion list for XPath expressions never includes axis names.

The default value for this option is `always`.

This command is equivalent to setting the `$XPATH_AXIS_COMPLETION` variable.

2.114. xpath-completion

Usage

```
xpath-completion expression
```

Aliases

```
xpath_completion
```

Description

If the expression is non-zero, enable the TAB completion for xpath expansions in the interactive shell mode, disable it otherwise. Defaults to on.

This command is equivalent to setting the `$XPATH_COMPLETION` variable.

2.115. xpath-extensions

Usage

```
xpath-extensions [expression]
```

Aliases

```
xpath_extensions
```

Description

`xpath-extensions` remaps all extra XPath extension functions provided by XSH2 (which by default belong to the namespace `http://xsh.sourceforge.net/xsh/`) to a new namespace given by expression. If the argument is omitted, the functions are re-registered without any namespace, so that they may be called without a namespace prefix, just by their function names. This quite convenient.

```
xpath-extensions;
ls grep(//word,"^.*tion$");
```

See Also

set-enc, set-standalone

2.116. xslt

Usage

```
$result := xslt [--doc|:d | --precompiled|:p] [--string] expression [document
name=xpath [name=xpath ...]]$pre_compiled := xslt [--compile|:c] expression
```

Aliases

```
transform, xsl, xsltproc, process
```

Description

This function compiles a given XSLT stylesheet and/or transforms a given document with XSLT.

A XSLT stylesheet is specified in the first argument either as a file name (default), or as a document (`--doc` or `:d`), or as a precompiled XSLT stylesheet object (`--precompiled` or `:p` - see `--compile` above).

If `--compile` or `:c` is used, compile a given XSLT stylesheet and return a compiled XSLT stylesheet object. This object can be later passed as a XSLT stylesheet to `xslt --precompiled`.

Without `--compile` or `:c`, transform a given document (or - if used with only the stylesheet argument - the current document) using a given XSLT stylesheet and return the result.

All arguments following the second (document) argument are considered to be stylesheet parameters and (after expanding `${...}` interpolators) are directly passed to the XSLT engine without being evaluated by XSH2. All stylesheet parameters should be of the form `name=xpath` (possibly in brackets).

Example 61. Process current document with XSLT

```
$result := xslt stylesheet.xsl . font='14pt' color='red'
```

Example 62. Same for several documents, reusing the XSLT stylesheet

```
$xslt := xslt --compile stylesheet.xsl;
foreach my $file in {qw(f1.xml f2.xml f3.xml)} {
  save --file {"out_$file"} &{xslt --precompiled $xslt &{ open $file } font='14pt'
color='red'};
}
```

2.117. xupdate

Usage

```
xupdate document [document]
```

Description

Modify the current document or the document specified by the second document argument according to XUpdate commands of the first document argument. XUpdate, or more specifically the XML Update Language, aims to be a language for updating XML documents.

XUpdate language is described in XUpdate Working Draft at http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html.

XUpdate output can be generated for example by Python xmldiff utility from http://www.logilab.org/projects/xmldiff/.

3. Type Reference

3.1. variable name

Description

Variable names start with a dollar-sign (`$`) followed by an identifier. The identifier must match the following regular expression `[a-zA-Z_][a-zA-Z0-9_]*`, i.e., it must be at least one character long, must beginning with a letter or underscore, and may only containing letters, underscores, and digits.

See Also

Variables, assign, my, local

3.2. filename

Description

An expression which evaluates to a valid filename or URL. As long as the expression contains no whitespace, no brackets of any type, quotes, double-quotes, `$` character nor `@` character, it is treated as a literal token which evaluates to itself.

3.3. node-name

Description

An expression which evaluates to a valid name of an element, attribute or processing-instruction node. As long as the expression contains no whitespace, no brackets of any type, quotes, double-quotes, `$` character, nor `@` character, it is treated as a literal token which evaluates to itself.

3.4. xPath

Description

XSH2 can evaluate XPath expressions as defined in W3C recommendation at http://www.w3.org/TR/xpath with only a little limitation on use of syntactically ignorable whitespace. (Nice interactive XPath tutorials and references can be found at http://www.zvon.org.)

In order to allow XSH2 to use white-space as a command argument delimiter (which is far more convenient to type than, say, commas), the use of white-space in XPath is slightly restricted. Thus, in XSH2, white-space can only occur in those parts of an XPath expression, that are surrounded by either brackets, square brackets, single or double quotes. So, for example, otherwise valid XPath expression like

```
/ foo / bar [ @baz = "bar" ]
```

should in XSH2 be written as either of

```
/foo/bar[ @baz = "bar" ]
```

avoiding any white-space outside the square brackets, or completely enclosed in brackets as in

```
( / foo / bar [ @baz = "bar" ] ).
```

XSH2 provides a number of powerful XPath extension functions, listed below and described in separate sections. XPath extension functions by default belong to XSH2 namespace `http://xsh.sourceforge.net/xsh/` with a namespace prefix set to `xsh`. A program may however call the xpath-extensions command to map XSH2 XPath extension functions into the default namespace, so that they may be used directly without any prefix.

XPath extension functions defined in XSH2: xsh:doc, xsh:filename, xsh:var, xsh:matches, xsh:match, xsh:grep, xsh:substr, xsh:reverse, xsh:lc, xsh:uc, xsh:lcfirst, xsh:ucfirst, xsh:same, xsh:max, xsh:min, xsh:strmax, xsh:strmin, xsh:sum, xsh:join, xsh:subst, xsh:sprintf, xsh:serialize, xsh:parse, xsh:current, xsh:path, xsh:if, xsh:new-attribute, xsh:new-element, xsh:new-element-ns, xsh:new-text, xsh:new-comment, xsh:new-pi, xsh:new-cdata, xsh:new-chunk, xsh:map, xsh:evaluate, xsh:split, xsh:times, xsh:id2, xsh:lookup, xsh:document, xsh:documents

Example 63. Open a document and count all sections containing a subsection

```
xsh $scratch/> $v := open mydocument1.xml;
xsh $v/> $k := open mydocument2.xml;
xsh $k/> count //section[subsection];# searches k
xsh $k/> count $v//section[subsection];# searches v
```

3.5. command

Description

XSH2 command consists of a command name and possibly command parameters separated by whitespace. Individual XSH2 commands are separated with a semicolon. A command may optionally be followed by an output redirection directive (see binding_shell for output redirection to a command and Variables for output redirection to variable). Most commands have aliases, so for example remove command may also be invoked as `del` or `rm`.

XSH2 recognizes the following commands (not including aliases): try, if, unless, while, do, eval, foreach, undef, def, assign, my, local, settings, defs, ifinclude, include, apropos, help, exec, xslt, documents, set_filename, variables, copy, xcopy, lcd, insert, wrap, wrap-span, xinsert, move, xmove, clone, normalize, strip-whitespace, ls, canonical, count, change-ns-uri, change-ns-prefix, set-ns, declare-ns, set, get, perl, remove, print, sort, map, rename, hash, close, index, open, create, save, set-dtd, dtd, set-enc, set-standalone, enc, validate, exit, process-xinclude, cd, pwd, locate, xupdate, verbose, test-mode, run-mode, debug, nodebug, version, validation, recovering, parser-expands-entities, keep-blanks, pedantic-parser, parser-completes-attributes, indent, empty-tags, skip-dtd, parser-expands-xinclude, load-ext-dtd, encoding, query-encoding, quiet, switch-to-new-documents, backups, nobackups, fold,

unfold, redo, next, prev, last, return, throw, catalog, iterate, register-namespace, unregister-namespace, register-xhtml-namespace, register-xsh-namespace, register-function, unregister-function, stream, namespaces, xpath-completion, xpath-axis-completion, doc-info, xpath-extensions, lineno, edit-string, edit, call

See Also

command-or-block

3.6. command-or-block

Description

XSH2 command or a block of semicolon-separated commands enclosed within braces.

Example 64. Count paragraphs in each chapter

```
$i=0;
foreach //chapter {
  $c=count(./para);
  $i=$i+1;
  print "$c paragraphs in chapter no.$i";
}
```

3.7. document

Description

A document is specified by arbitrary expression which evaluates to a non-empty node-list. From this node-list, the first node is taken and its owner document is used.

See Also

Variables, assign, my, local

3.8. enc_string

Description

An expression which evaluates to a valid encoding name, e.g. to utf-8, utf-16, iso-8859-1, iso-8859-2, windows-1250 etc. As with filename, as long as the expression doesn't contain special characters like braces, brackets, quotes, `$`, nor `@`, it is taken as a literal and evaluates to itself.

3.9. expression

Description

An XSH2 expression can be one of the following constructs:

1. XPath 1.0 expression with the following restriction: whitespace is only allowed within parts the expression enclosed in quotes (literal strings) or brackets (XPath has two types of brackets - plain and square). Thus, while `/ foo / bar` is a valid XPath expression matching element named bar under root element foo, in XSH2 this expression must be written as `/foo/bar` or `(/ foo / bar)` or `(/foo/bar)` etc. The reason for this restriction is simple: XSH2, like most shell languages, uses whitespace as argument delimiter so it must be able to determine expression boundaries (otherwise, `/ bar / foo` could be anything between one and four expressions).

2. In certain contexts, usually when a filename or a node name is expected as an argument, bareword (otherwise XPath) expressions are evaluated in a non-standard way: as long as the expression contains no whitespace, no brackets of any kind, quotes, double-quotes, `$` character, nor `@` character, it is treated as a literal token which evaluates to itself. This usually happens if a file name or element name is expected, but some other commands, like print, evaluate its arguments in this way. In order to force an XPath evaluation in such situations, the entire expression should be enclosed with brackets `(...)`. For example, with open command, `open file` or `open "file"` both open a file whose name is `file` (literally) whereas `open (file)` or `open @file` compute the file name by evaluating `(file)` or `@file` respectively, as XPath expressions.

3. Perl blocks. These are enclosed in braces like: `{ perl-code }`. Perl expressions can be used to evaluate more complicated things, like complex string expressions, regexp matches, perl commands, etc. In short, arbitrary perl. Of course, things like `{`ls`}` work too, and that's why we don't need to define shell-like backticks in XSH2 itself.

4. Result of one XSH2 command can be directly passed as an argument to another. This is done using &{ xsh-code } expressions. Most XSH2 commands always return `undef` or 1, but some do return a value, usually a node-list. Examples of such commands are open, copy, move, wrap, edit, or xslt.

5. Large blocks of literal data can be passed to commands via "here document" expressions <<EOF, <<'EOF', <<"EOF", where `EOF` is an arbitrary `ID` string. <<EOF and <<"EOF" are equivalent, and are subject to interpolation of `${...}` constructs, where as <<'EOF' does not. The result of evaluation of these three is the literal content (with `${...}` possibly interpolated) of the script starting at the following line and ending at a line containing just `EOF`. <<{EOF} and <<(EOF) are implemented too, but I'm not sure they are of any use since putting the expression in ( ) or { } has the same effect.

XPath expressions (and their filename variant) are subject to interpolation of substrings of the form `${...}` (called interpolators), where `...` can be of several different forms, described below. The interpolation can be suppressed by preceding the `$` sign with a backslash.

Substrings of the form `${id}` or `${$id}` are interpolated with the value of the variable named `$id`.

Interpolators of the form `${{` and `}}` evaluate their contents as a Perl expression (in very much the same way as the perl command) and interpolate to the resulting value.

Interpolators of the form `${(` and `)}` evaluate their contents as an XPath expression and interpolates to a string value of the result.

Substrings of the form `\${` interpolate to `${` (as a means for escaping `${...}` in an expression).

Expressions are evaluated by XSH2 commands themselves, so the exact value an expression evaluates to, is also command-dependent. There are commands that can handle all data types, but some commands expect their arguments to evaluate only to specific kinds of values. As already mentioned above, commands expecting a filename or a node name usually evaluate simple expressions not containing any special characters as literal strings, whereas commands expecting strings evaluate all expressions so that they get a string value (e.g. by converting a node-set to its text content). Similarly, commands expecting a node-set usually convert strings to a small XML fragments, while commands expecting a single document node usually convert node-sets to a document node by taking the owner document of the first element in the node-set.

```
$a = "bar";               # $a contains: bar

$b = $a;                  # $b contains: bar

$b = "$a";                # $b contains: $a

$b = "${a}";              # $b contains: bar

$b = {$a};                # $b contains: bar

$b = //creature;          # $b contains a node-set

ls $b;                    # prints the node-set as XML in document order
```

```
count $b;                 # prints number of nodes in the node-set

echo count($b);           # the same

$c = string($b[1]/@name) # $c contains string value of //creature[1]/@name (e.g.
Bilbo)

echo //creature           # prints: //creature

echo (//creature)         # evaluates (//creature) as XPath and prints the
# text content of the resulting node-set

echo { join(",",split(//,$a)) }              # prints: b,a,r

echo ${{ join(",",split(//,$a)) }}           # the same

echo "${{ join(",",split(//,$a)) }}"         # the same

echo "${(//creature[1]/@name)}"              # prints e.g.: Bilbo

echo ${(//creature[1]/@name)}                # the same

echo //creature[1]/@name                     # the same

echo string(//creature[1]/@name)             # the same

echo (//creature[1]/@name)                   # the same
```

Example 65. In-line documents

```
$a="bar"

echo foo <<END baz;
xx ${a} yy
END
# prints foo xx bar yy baz

echo foo <<"END" baz;
xx ${a} yy
END
# same as above

echo foo <<'END' baz;
xx ${a} yy
END
# prints foo xx $a yy baz
```

Example 66. Expressions returning result of a XSH2 command

```
copy &{ sort --key @best_score --numeric //player } into .;
```

3.10. location

Description

One of: `after`, `before`, `into`, `append`, `prepend`, `replace`.

This argument is required by all commands that insert nodes to a document in some way to a destination described by an XPath expression. The meaning of the values listed above is supposed be obvious in most cases, however the exact semantics for location argument values depends on types of both the source node and the target node.

`after/before` place the node right after/before the destination node, except for when the destination node is a document node or one of the source nodes is an attribute: If the destination node is a document node, the source node is attached to the end/beginning of the document (remember: there is no "after/before a document"). If both

the source and destination nodes are attributes, then the source node is simply attached to the element containing the destination node (remember: there is no order on attribute nodes). If the destination node is an attribute but the source node is of a different type, then the textual content of the source node is appended to the value of the destination attribute (i.e. in this case after/before act just as append/prepend).

`append/prepend` appends/prepends the source node to the destination node. If the destination node can contain other nodes (i.e. it is an element or a document node) then the entire source node is attached to it. In case of other destination node types, the textual content of the source node is appended/prepended to the content of the destination node.

`into` can also be used to place the source node to the end of an element (in the same way as `append`), to attach an attribute to an element, or, if the destination node is a text node, cdata section, processing-instruction, attribute or comment, to replace its textual content with the textual content of the source node.

`replace` replaces the entire destination node with the source node except for the case when the destination node is an attribute and the source node is not. In such a case only the value of the destination attribute is replaced with the textual content of the source node. Note also that document node can never be replaced.

3.11. node-type

Description

One of: element, attribute, text, cdata, comment, chunk and (EXPERIMENTALLY!) entity_reference. A chunk is a character string which forms a well-balanced piece of XML.

```
add element hobbit into //middle-earth/creatures;
add attribute 'name="Bilbo"' into //middle-earth/creatures/hobbit[last()];
add chunk '<hobbit name="Frodo">A small guy from <place>Shire</place>.</hobbit>'
  into //middle-earth/creatures;
```

3.12. perl-code

Description

A block of Perl code enclosed in braces. All XSH2 variables are transparently accessible from the Perl code as well.

For more information about embedded Perl code in XSH2, predefined functions etc., see Perl_shell.

```
xsh> $i={ "foo" };
xsh> perl { echo "$i-bar\n"; }# prints foo-bar
xsh> echo { "$i-bar" }# very much the same as above
```

3.13. sub-routine name

Description

A sub-routine name is an identifier matching the following regular expression `[a-zA-Z_][a-zA-Z0-9_]*`, i.e., it must be at least one character long, must beginning with a letter or underscore, and may only containing letters, underscores, and digits.

4. XPath Extension Function Reference

4.1. xsh:current

Usage

```
node-set xsh:current()
```

Description

This function (very similar to XSLT `current()` extension function) returns a node-set having the current node as its only member.

4.2. xsh:doc

Usage

```
node-set xsh:doc(node-set)
```

Description

Returns a node-set consisting of the owner document nodes of all nodes in the given node-set.

4.3. xsh:document

Usage

```
node-set xsh:document(string URL)
```

Description

Looks up among the currently open document the one whose filename is same as the given URL and returns the corresponding document node. If no document's filename matches exactly the given URL, then several heuristic matches are tried: if the URI is a relative filename, it is tilde-expanded and resolved (using the current working directory as a base) and the lookup is restarted with the absolute filename; finally, a lookup identifying filenames with URLs of the file:// protocol is attempted. If the lookup fails completely, an empty node set is returned.

See Also

hash

4.4. xsh:documents

Usage

```
node-set xsh:documents()
```

Description

Returns a node-set consisting of the document nodes of all currently open documents.

See Also

hash

4.5. xsh:evaluate

Usage

```
node-set xsh:evaluate(string XPATH)
```

Description

This function is very similar to EXSLT `dynamic:evaluate` function. The description below is almost literally taken from the EXSLT specification [http://www.exslt.org/dyn/functions/map/index.html].

The `xsh:evaluate` function evaluates a string as an XPath expression and returns the resulting value, which might be a boolean, number, string, node set, result tree fragment or external object. The sole argument is the string to be evaluated.

The string is always evaluated exactly as if it had been literally included in place of the call to the `xsh:evaluate` function.

In other words, the context information used when evaluating the XPath expression passed as the argument to the `xsh:evaluate` function is exactly the same as the context information used when evaluating the `xsh:evaluate` function. This context information includes:

1. the context node, such that paths are evaluated relative to the context node at the point where the `xsh:evaluate` function is called
2. the context position, such that the expression can contain calls to the position function
3. the context size, such that the expression can contain calls to the last function
4. variable bindings, such that the expression can contain variable references
5. function library, such that the expression can contain calls to extension functions
6. namespace declarations, such that paths can contain prefixes the current node, such that the expression can contain calls to the current function

If the expression string passed as the second argument is an invalid XPath expression (including an empty string), this function returns an empty node set.

You should only use this function if the expression must be constructed dynamically - otherwise it is much more efficient to use the expression literally. For expressions that simply give an element or attribute's name (to select a child element or attribute), it is more efficient to use an expression in the style:

```
*[name() = $expression]
```

See Also

xsh:map

4.6. xsh:filename

Usage

```
node-set xsh:filename(node-set?)
```

Description

Returns filename (URL) of the document containing the first node in the given node-set. If called without arguments, or if the node-set is empty, returns filename of the document containing the current node.

4.7. xsh:grep

Usage

```
node-set xsh:grep(node-set NODES, string PATTERN)
```

Description

Returns a node set consisting of those nodes of `NODES` whose content (as returned by the built-in XPath function `string()`) matches the regular expression `PATTERN`.

4.8. xsh:id2

Usage

```
node-set xsh:id2(node-set DOC, string IDs)
```

Description

This function is like XPath built-in `id(IDs)` function, except that it operates on the document specified in the first argument. It returns a node-set consisting of nodes that belong to the document DOC and whose ID belongs to the list of space separated IDs specified in the second argument.

4.9. xsh:if

Usage

```
object xsh:if(object CONDITION, object YES, object NO)
```

Description

This function returns the `YES` object if `CONDITION` is an non-empty node-set or a string, boolean or integer evaluating to non-zero boolean. Otherwise the `NO` object is returned.

4.10. xsh:join

Usage

```
string xsh:join(string DELIM, object EXPRESSION,...)
```

Description

Joins the separate string values computed from `EXPRESSION`(s) into a single string with fields separated by the value of `DELIM`, and returns that new string. If `EXPRESSION` evaluates to a node-set, joins string values of individual nodes.

4.11. xsh:lc

Usage

```
string xsh:lc(string STR)
```

Description

Returns a lowercased version of `STR`.

4.12. xsh:lcfirst

Usage

```
string xsh:lcfirst(string STR)
```

Description

Returns the value of `STR` with the first character lowercased.

4.13. xsh:lookup

Usage

```
node-set xsh:lookup(string VARNAME, string KEY)
```

Description

This function is similar to XSLT `key()` function. It returns a node-set stored in a hash VARNAME under the key KEY. The VARNAME must be a name of a lexical or global XSH variable containing a Perl hash reference.

See Also

hash

4.14. xsh:map

Usage

```
node-set xsh:map(node-set NODE, string XPATH)
```

Description

This function is very similar to EXSLT `dynamic:map` function. The description below is almost literally taken from the EXSLT specification [http://www.exslt.org/dyn/functions/map/index.html].

The `xsh:map` function evaluates the expression passed as the second argument for each of the nodes passed as the first argument, and returns a node-set of those values.

The expressions are evaluated relative to the nodes passed as the first argument. In other words, the value for each node is calculated by evaluating the XPath expression with all context information being the same as that for the call to the `xsh:map` function itself, except for the following:

1) the context node is the node whose value is being calculated, 2) the context position is the position of the node within the node set passed as the first argument to the `xsh:map` function, arranged in document order, and 3) the context size is the number of nodes passed as the first argument to the dyn:map function.

If the expression string passed as the second argument is an invalid XPath expression (including an empty string), this function returns an empty node set.

If `XPATH` evaluates as a node set, the `xsh:map` function returns the union of the node sets returned by evaluating the expression for each of the nodes in the first argument. Note that this may mean that the node set resulting from the call to the `xsh:map` function contains a different number of nodes from the number in the node set passed as the first argument to the function.

If `XPATH` evaluates as a number, the `xsh:map` function returns a node set containing one `xsh:number` element (namespace `http://xsh.sourceforge.net/xsh/`) for each node in the node set passed as the first argument to the dyn:map function, in document order. The string value of each `xsh:number` element is the same as the result of converting the number resulting from evaluating the expression to a string as with the number function, with the exception that Infinity results in an `xsh:number` holding the highest number the implementation can store, and -Infinity results in an `xsh:number` holding the lowest number the implementation can store.

If `XPATH` evaluates as a boolean, the `xsh:map` function returns a node set containing one `xsh:boolean` element (namespace `http://xsh.sourceforge.net/xsh/`) for each node in the node set passed as the first argument to the `xsh:map` function, in document order. The string value of each `xsh:boolean` element is `true` if the expression evaluates as true for the node, and is empty if the expression evaluates as false.

Otherwise, the `xsh:map` function returns a node set containing one `xsh:string` element (namespace `http://xsh.sourceforge.net/xsh/`) for each node in the node set passed as the first argument to the `xsh:map` function, in document order. The string value of each `xsh:string` element is the same as the result of converting the result of evaluating the expression for the relevant node to a string as with the string function.

See Also

xsh:evaluate

4.15. xsh:match

Usage

```
boolean xsh:match(string STR,string PATTERN,options STR)
```

Description

Searches a given string for a pattern match specified by a regular expression `PATTERN` and returns a node-set consisting of `<xsh:string>` elements containing portions of the string matched by the pattern subexpressions enclosed in parentheses.

4.16. xsh:matches

Usage

```
boolean xsh:matches(string STR,string PATTERN)
```

Description

Returns `true` if `STR` matches the regular expression `PATTERN`. Otherwise returns `false`.

4.17. xsh:max

Usage

```
float xsh:max(object EXPRESSION, ...)
```

Description

Returns the maximum of numeric values computed from given `EXPRESSION`(s). If `EXPRESSION` evaluates to a node-set, string values of individual nodes are used.

4.18. xsh:min

Usage

```
float xsh:min(object EXPRESSION, ...)
```

Description

Returns the minimum of numeric values computed from given `EXPRESSION`(s). If `EXPRESSION` evaluates to a node-set, string values of individual nodes are used.

4.19. xsh:new-attribute

Usage

```
node-set xsh:new-attribute(string NAME1,string VALUE1,[string NAME2, string
VALUE2, ...])
```

Description

Return a node-set consisting of newly created attribute nodes with given names and respective values.

4.20. xsh:new-cdata

Usage

```
node-set xsh:new-cdata(string DATA)
```

Description

Create a new cdata section node node filled with given `DATA` and return a node-set containing the new node as its only member.

4.21. xsh:new-chunk

Usage

```
node-set xsh:new-chunk(string XML)
```

Description

This is just an alias for xsh:parse. It parses given piece of XML and returns a node-set consisting of the top-level element within the parsed tree.

4.22. xsh:new-comment

Usage

```
node-set xsh:new-comment(string DATA)
```

Description

Create a new comment node containing given `DATA` and return a node-set containing the new node as its only member.

4.23. xsh:new-element

Usage

```
node-set xsh:new-element(string NAME,[string ATTR1-NAME1, string ATTR-VALUE1,
...])
```

Description

Create a new element node with given `NAME` and optionally attributes with given names and values and return a node-set containing the new node as its only member.

4.24. xsh:new-element-ns

Usage

```
node-set xsh:new-element-ns(string NAME,string NS,[string ATTR1-NAME1, string
ATTR-VALUE1, ...])
```

Description

Create a new element node with given `NAME` and namespace-uri `NS` and optionally attributes with given names and values and return a node-set containing the new node as its only member.

4.25. xsh:new-pi

Usage

```
node-set xsh:new-pi(string NAME, [string DATA])
```

Description

Create a new processing instruction node node with given `NAME` and (optionally) given `DATA` and return a node-set containing the new node as its only member.

4.26. xsh:new-text

Usage

```
node-set xsh:new-text(string DATA)
```

Description

Create a new text node containing given `DATA` and return a node-set containing the new node as its only member.

4.27. xsh:parse

Usage

```
node-set xsh:parse(string XML-STRING)
```

Description

This function runs XML parser on `XML-STRING` and returns a node-set consisting of the top-level nodes of the resulting document node.

4.28. xsh:path

Usage

```
string xsh:path(node-set NODE)
```

Description

This function returns a string containing canonical XPath leading to `NODE`.

See Also

pwd

4.29. xsh:reverse

Usage

```
string xsh:reverse(string STR)
```

Description

Returns a string value same as `STR` but with all characters in the opposite order.

4.30. xsh:same

Usage

```
bool xsh:same(node-set N1, node-set N2)
```

Description

Returns `true` if the given node sets both contain the same node (in XPath, this can also be expressed as `count(N1|N2)+count(N1)+count(N2)=3`).

4.31. xsh:serialize

Usage

```
string xsh:serialize(node-set N,...)
```

Description

Serializes nodes of given node-set(s) into XML strings and returns concatenation of those strings.

4.32. xsh:split

Usage

```
node-set xsh:split(string PATTERN, string STRING)
```

Description

This function provides direct access to the very powerful Perl function `split`. It splits `STRING` to a list of fields. `PATTERN` is a regular expression specifying strings delimiting individual fields of `STRING`. If `PATTERN` is empty, `STRING` is split to individual characters. If the regular expression in `PATTERN` is enclosed in brackets, then strings matching `PATTERN` are also included in the resulting list.

The function returns a node-set consisting of newly created `<xsh:string>` elements containing individual strings of the resulting list as their only text child nodes.

4.33. xsh:sprintf

Usage

```
string xsh:sprintf(string FORMAT,object EXPRESSION,...)
```

Description

Returns a string formatted by the usual `printf` conventions of the C library function `sprintf` and `sprintf` Perl function.

See C documentation for an explanation of the general principles and Perl documentation for a list of supported formatting conversions.

4.34. xsh:strmax

Usage

```
string xsh:strmax(object EXPRESSION, ...)
```

Description

Returns a string value computed as the maximum (in lexicographical order) of all string values computed from given EXPRESSION(s). If EXPRESSION evaluates to a node-set, string values of individual nodes are used.

4.35. xsh:strmin

Usage

```
string xsh:strmin(object EXPRESSION, ...)
```

Description

Returns a string value computed as the minimum (in lexicographical order) of all string values computed from given EXPRESSION(s). If EXPRESSION evaluates to a node-set, string values of individual nodes are used.

4.36. xsh:subst

Usage

```
string xsh:subst(string STR,string REGEXP,string REPLACEMENT, [string OP-
TIONS])
```

Description

Acts in the very same way as perl substitution operation STRING =~ s/REGEXP/REPLACEMENT/OPTIONS, returning the resulting string. Searches a string for a pattern, and if found, replaces that pattern with the replacement text. If the REPLACEMENT string contains a $ that looks like a variable, the variable will be interpolated into the

REPLACEMENT at run-time. Options are:

e - evaluate REPLACEMENT as a Perl expression,

g - replace globally, i.e., all occurrences,

i - do case-insensitive pattern matching,

m - treat string as multiple lines, that is, change ^ and $ from matching the start or end of the string to matching the start or end of any line anywhere within the string,

s - treat string as single line, that is, change . to match any character whatsoever, even a newline, which normally it would not match,

x - use extended regular expressions.

4.37. xsh:substr

Usage

```
string xsh:substr(string STR,float OFFSET,[float LENGTH])
```

Description

Extracts a substring out of STR and returns it. First character is at offset 0.

If OFFSET is negative, starts that far from the end of the string.

If LENGTH is omitted, returns everything to the end of the string. If LENGTH is negative, leaves that many characters off the end of the string.

If OFFSET and LENGTH specify a substring that is partly outside the string, only the part within the string is returned. If the substring is beyond either end of the string, substr() returns empty string and produces a warning.

4.38. xsh:sum

Usage

```
float xsh:sum(object EXPRESSION, ...)
```

Description

Returns the sum of numerical value computed from given EXPRESSION(s). If EXPRESSION evaluates to a node-set, string values of individual nodes are used.

4.39. xsh:times

Usage

```
node-set xsh:times(string STRING, float COUNT)
```

Description

This function returns a string resulting from concatenation of COUNT copies of STRING. COUNT must be a non-negative integer value.

4.40. xsh:uc

Usage

```
string xsh:uc(string STR)
```

Description

Returns a uppercased version of STR.

4.41. xsh:ucfirst

Usage

```
string xsh:ucfirst(string STR)
```

Description

Returns the value of STR with the first character uppercased.

4.42. xsh:var

Usage

```
node-set xsh:var(string NAME)
```

Description

Returns a node-set consisting of nodes stored in a XSH2 node-list variable named `NAME`.