NAME

XML::XPath - Parse and evaluate XPath statements.

VERSION

Version 1.40

DESCRIPTION

This module aims to comply exactly to the XPath specification at http://www.w3.org/TR/xpath and yet allow extensions to be added in the form of functions.Modules such as XSLT and XPointer may need to do this as they support functionality beyond XPath.

SYNOPSIS

```
use XML::XPath;
use XML::XPath::XMLParser;

my $xp = XML::XPath->new(filename => 'test.xhtml');

my $nodeset = $xp->find('/html/body/p'); # find all paragraphs

foreach my $node ($nodeset->get_nodelist) {
print "FOUND\n\n",
XML::XPath::XMLParser::as_string($node),
"\n\n";
}
```

DETAILS

There is an awful lot to all of this, so bear with it - if you stick it out it should be worth it. Please get a good understanding of XPath by reading the spec before asking me questions. All of the classes and parts herein are named to be synonymous with the names in the specification, so consult that if you don't understand why I'm doing something in the code.

METHODS

The API of XML::XPath itself is extremely simple to allow you to get going almost immediately. The deeper API's are more complex, but you shouldn't have to touch most of that.

new()

This constructor follows the often seen named parameter method call. Parameters you can use are: filename, parser, xml, ioref and context. The filename parameter specifies an XML file to parse. The xml parameter specifies a string to parse, and the ioref parameter specifies an ioref to parse. The context option allows you to specify a context node. The context node has to be in the format of a node as specified in XML::XPath::XMLParser. The 4 parameters filename, xml, ioref and context are mutually exclusive - you should only specify one (if you specify anything other than context, the context node is the root of your document). The parser option allows you to pass in an already prepared XML::Parser object, to save you having to create more than one in your application (if, for example, you are doing more than just XPath).

```
my $xp = XML::XPath->new( context => $node );
```

It is very much recommended that you use only 1 XPath object throughout the life of your application. This is because the object (and it's sub-objects) maintain certain bits of state information that will be useful (such as XPath variables) to later calls to *find()*. It's also a good idea because you'll use less memory this way.

find($path, [$context])

The find function takes an XPath expression (a string) and returns either an XML::XPath::NodeSet object containing the nodes it found (or empty if no nodes matched the path), or one of XML::XPath::Literal (a string), XML::XPath::Number or XML::XPath::Boolean. It should always return something - and you can use ->*isa()* to find out what it returned. If you need to check how many nodes it found you should check `$nodeset`->size. See XML::XPath::NodeSet. An optional second parameter of a context node allows you to use this method repeatedly, for example XSLT needs to do this.

findnodes($path, [$context])
Returns a list of nodes found by `$path`, optionally in context `$context`. In scalar context returns an XML::XPath::NodeSet object.

matches($node,** `$path`**, [$context])
Returns true if the node matches the path (optionally in context `$context`).

findnodes_as_string($path, [$context])
Returns the nodes found reproduced as XML.The result isn't guaranteed to be valid XML though.

findvalue($path, [$context])
Returns either a `XML::XPath::Literal` a `XML::XPath::Boolean` or a `XML::XPath::Number` object.If the path returns a NodeSet,$nodeset->to_literal is called automatically for you (and thus a `XML::XPath::Literal` is returned).Note that for each of the objects stringification is overloaded, so you can just print the value found, or manipulate it in the ways you would a normal perl value (e.g. using regular expressions).

exists($path, [$context])
Returns true if the given path exists.

getNodeText($path)
Returns the XML::XPath::Literal for a particular XML node. Returns a string if exists or '' (empty string) if the node doesn't exist.

setNodeText($path, `$text`**)**
Sets the text string for a particular XML node. The node can be an element or an attribute. If the node to be set is an attribute, and the attribute node does not exist, it will be created automatically.

createNode($path)
Creates the node matching the `$path` given. If part of the path given or all of the path do not exist, the necessary nodes will be created automatically.

set_namespace($prefix, `$uri`**)**
Sets the namespace prefix mapping to the uri.

Normally in `XML::XPath` the prefixes in XPath node test take their context from the current node. This means that foo:bar will always match an element <foo:bar> regardless of the namespace that the prefix foo is mapped to (which might even change within the document, resulting in unexpected results). In order to make prefixes in XPath node tests actually map to a real URI, you need to enable that via a call to the set_namespace method of your `XML::XPath` object.

clear_namespaces()
Clears all previously set namespace mappings.

`$XML::XPath::Namespaces`
Set this to 0 if you *don't* want namespace processing to occur. This will make everything a little (tiny) bit faster, but you'll suffer for it, probably.

Node Object Model

See XML::XPath::Node, XML::XPath::Node::Element, XML::XPath::Node::Text, XML::XPath::Node::Comment, XML::XPath::Node::Attribute, XML::XPath::Node::Namespace, and XML::XPath::Node::PI.

On Garbage Collection

XPath nodes work in a special way that allows circular references, and yet still lets Perl's reference counting garbage collector to clean up the nodes after use. This should be totally transparent to the user, with one caveat: **If you free your tree before letting go of a sub-tree,consider that playing with fire and you may get burned**. What does this mean to the average user? Not much. Provided you don't free (or let go out of scope) either the tree you passed to XML::XPath->new, or if you didn't pass a tree, and passed a filename or IO-ref, then provided you don't let the XML::XPath object go out of scope before you let results of *find()* and its friends go out of scope, then you'll be fine. Even if you **do** let the tree go out of scope before results, you'll probably still be fine. The only case where you may get stung is when the last part of your path/query is either an ancestor or parent axis. In that case the worst that will happen is you'll end up with a circular reference that won't get cleared until interpreter destruction time.You can get around that by explicitly calling `$node`->DESTROY on each of your result nodes, if you really need to do that.

Mail me direct if that's not clear. Note that it's not doom and gloom. It's by no means perfect,but the worst that will happen is a long running process could leak memory. Most long running processes will therefore be able to explicitly be careful not to free the tree (or XML::XPath object) before freeing results.AxKit, an application that uses XML::XPath, does this and I didn't have to make any changes to the code - it's already sensible programming.

If you *really* don't want all this to happen, then set the variable `$XML::XPath::SafeMode`, and call `$xp`->*cleanup()* on the XML::XPath object when you're finished, or `$tree`->*dispose()* if you have a tree instead.

Example

Please see the test files in t/ for examples on how to use XPath.

AUTHOR

Original author Matt Sergeant, `<matt at sergeant.org>`

Currently maintained by Mohammad S Anwar, `<mohammad.anwar at yahoo.com>`

SEE ALSO

XML::XPath::Literal, XML::XPath::Boolean, XML::XPath::Number, XML::XPath::XMLParser, XML::XPath::NodeSet, XML::XPath::PerlSAX, XML::XPath::Builder.

LICENSE AND COPYRIGHT

For support, please subscribe to the Perl-XML <http://listserv.activestate.com/mailman/listinfo/perl-xml> mailing list at the URL