Package: XML (via r-universe)

October 28, 2026

Version 3.99-0.25

Title Tools for Parsing and Generating XML Within R and S-Plus

Depends R (>= 4.0.0), methods, utils

Suggests bitops, RCurl

SystemRequirements libxml2 (>= 2.6.3)

Description Many approaches for both reading and creating XML (and HTML) documents, both local and accessible via HTTP or FTP. Also offers access to an 'XPath' ``interpreter".

BugReports https://codeberg.org/aitap/XML/issues

License BSD_3_clause + file LICENSE

Collate AAA.R DTDClasses.R DTDRef.R SAXMethods.R XMLClasses.R applyDOM.R assignChild.R catalog.R createNode.R dynSupports.R error.R flatTree.R nodeAccessors.R schema.R summary.R tangle.R toString.R tree.R version.R xmlErrorEnums.R xmlEventHandler.R xmlEventParse.R xmlHandler.R xmlInternalSource.R xmlOutputDOM.R xmlNodes.R xmlOutputBuffer.R xmlTree.R xmlTreeParse.R htmlParse.R hashTree.R zzz.R supports.R parser.R libxmlFeatures.R xmlString.R saveXML.R namespaces.R readHTMLTable.R reflection.R xmlToDataFrame.R bitList.R compare.R encoding.R fixNS.R xmlRoot.R serialize.R xmlMemoryMgmt.R keyValueDB.R solrDocs.R XMLRErrorInfo.R xincludes.R namespaceHandlers.R tangle1.R htmlLinks.R htmlLists.R getDependencies.R getRelativeURL.R xmlIncludes.R simplifyPath.R defunct.R

NeedsCompilation yes

Author CRAN Team [ctb] (de facto maintainer in 2013-2026), Duncan Temple Lang [aut] (ORCID: <https://orcid.org/0000-0003-0159-1546>), Tomas Kalibera [ctb], Ivan Krylov [cre]

Maintainer Ivan Krylov <ikrylov@disroot.org>

Config/pak/sysreqs libxml2-dev

Repository https://cran.r-universe.dev

Date/Publication 2026-09-27 23:40:02 UTC

RemoteUrl https://github.com/cran/XML

RemoteRef HEAD

RemoteSha c35ad6cc95e4bc43d42eaeff14d957a083a87e0d

Contents

[.XMLNode 4
[<-.XMLNode 5
addChildren 6
addNode 10
append.xmlNode 11
asXMLNode 13
asXMLTreeNode 14
catalogLoad 15
catalogResolve 17
coerceNodes 18
compareXMLDocs 19
docName 20
Doctype 21
Doctype-class 22
ensureNamespace 23
findXInclude 24
free 25
genericSAXHandlers 26
getChildrenStrings 27
getEncoding 28
getHTMLLinks 29
getLineNumber 31
getNodeSet 32
getRelativeURL 40
getSibling 41
getXIncludes 43
getXMLErrors 44
isXMLString 45
length.XMLNode 46
libxmlVersion 47
makeClassTemplate 48
names.XMLNode 50
newXMLDoc 51
newXMLNamespace 57
parseURI 58
parseXMLAndAdd 59
print.XMLAttributeDef 61
processXInclude 63
readHTMLList 64
readHTMLTable 65

readKeyValueDB . . . 68
readSolrDoc . . . 69
removeXMLNamespaces . . . 70
replaceNodeWithChildren . . . 71
saveXML . . . 72
SAXState-class . . . 74
schema-class . . . 76
setXMLNamespace . . . 77
startElement.SAX . . . 78
supportsExpat . . . 79
toHTML . . . 80
toString.XMLNode . . . 81
xmlApply . . . 82
XMLAttributes-class . . . 83
xmlAttributeType . . . 84
xmlAttrs . . . 85
xmlChildren . . . 86
xmlCleanNamespaces . . . 87
xmlClone . . . 88
XMLCodeFile-class . . . 89
xmlDOMApply . . . 91
xmlElementsByTagName . . . 92
xmlElementSummary . . . 94
xmlEventHandler . . . 95
xmlEventParse . . . 96
xmlGetAttr . . . 104
xmlHandler . . . 105
xmlHashTree . . . 106
XMLInternalDocument-class . . . 109
xmlName . . . 110
xmlNamespace . . . 111
xmlNamespaceDefinitions . . . 112
xmlNode . . . 114
XMLNode-class . . . 116
xmlOutputBuffer . . . 117
xmlParent . . . 119
xmlParseDoc . . . 121
xmlParserContextFunction . . . 123
xmlRoot . . . 124
xmlSchemaValidate . . . 125
xmlSearchNs . . . 127
xmlSerializeHook . . . 128
xmlSize . . . 129
xmlSource . . . 130
xmlStopParser . . . 133
xmlStructuredStop . . . 135
xmlToDataFrame . . . 136
xmlToList . . . 138

xmlToS4 139
xmlTree 140
xmlTreeParse 143
xmlValue 153

Index **156**

`[.XMLNode` *Convenience accessors for the children of XMLNode objects.*

Description

These provide a simplified syntax for extracting the children of an XML node.

Usage

```
## S3 method for class 'XMLNode'
x[..., all = FALSE]
## S3 method for class 'XMLNode'
x[[...]]
## S3 method for class 'XMLDocumentContent'
x[[...]]
```

Arguments

`x`	the XML node or the top-level document content in which the children are to be accessed. The `XMLDocumentContent` is the container for the top-level node that also contains information such as the URI/filename and XML version. This accessor method is merely a convenience to get access to children of the top-level node.
`...`	the identifiers for the children to be retrieved, given as integer indices, names, etc. in the usual format for the generic `link{[}` and `link{[[}` operators
`all`	logical value. When `...` is a character vector, a value of `TRUE` for `all` means to retrieve all of the nodes with those names rather than just the first one. `FALSE` gives the usual result of subsetting a list by name which gives just the first element. This allows us to avoid the idiom `node[ names(node) == "bob" ]` which is complicated when node is the result of an inline computation and instead we use `node["bob", all = TRUE]`.

Value

A list or single element containing the children of the XML node given by `obj` and identified by `....`

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

xmlAttrs [<-.XMLNode [[<-.XMLNode

Examples

```
f = system.file("exampleData", "gnumeric.xml", package = "XML")

top = xmlRoot(xmlTreeParse(f))

 # Get the first RowInfo element.
top[["Sheets"]][[1]][["Rows"]][["RowInfo"]]

 # Get a list containing only the first row element
top[["Sheets"]][[1]][["Rows"]]["RowInfo"]
top[["Sheets"]][[1]][["Rows"]][1]

 # Get all of the RowInfo elements by position
top[["Sheets"]][[1]][["Rows"]][1:xmlSize(top[["Sheets"]][[1]][["Rows"]])]

 # But more succinctly and accurately, get all of the RowInfo elements
top[["Sheets"]][[1]][["Rows"]]["RowInfo", all = TRUE]
```

[<-.XMLNode *Assign sub-nodes to an XML node*

Description

These functions allow one to assign a sub-node to an existing XML node by name or index. These are the assignment equivalents of the subsetting accessor functions. They are typically called indirectly via the assignment operator, such as `x[["myTag"]] <- xmlNode("mySubTag")`.

Usage

```
## S3 replacement method for class 'XMLNode'
x[i] <- value
## S3 replacement method for class 'XMLNode'
x[[i]] <- value
```

Arguments

x	the XMLNode object to which the sub-node is to be assigned.
i	the identifier for the position in the list of children of x into which the right-hand-side node(s) should be assigned. These can be either numbers or names.
value	one or more XMLNode objects which are to be the sub-nodes of x.

Value

The XML node x containing the new or modified nodes.

Author(s)

Duncan Templle Lang

References

https://www.w3.org

See Also

[.XMLNode [[.XMLNode append.xmlNode xmlSize

Examples

```
top <- xmlNode("top", xmlNode("next","Some text"))
top[["second"]] <- xmlCDataNode("x <- 1:10")
top[[3]] <- xmlNode("tag",attrs=c(id="name"))
```

---

addChildren *Add child nodes to an XML node*

Description

This collection of functions allow us to add, remove and replace children from an XML node and also to and and remove attributes on an XML node. These are generic functions that work on both internal C-level XMLInternalElementNode objects and regular R-level XMLNode objects.

addChildren is similar to addNode and the two may be consolidated into a single generic function and methods in the future.

Usage

```
addChildren(node, ..., kids = list(...), at = NA, cdata = FALSE, append = TRUE)
removeChildren(node, ..., kids = list(...), free = FALSE)
removeNodes(node, free = rep(FALSE, length(node)))
replaceNodes(oldNode, newNode, ...)
addAttributes(node, ..., .attrs = NULL,
          suppressNamespaceWarning = getOption("suppressXMLNamespaceWarning", FALSE),
                append = TRUE)
removeAttributes(node, ..., .attrs = NULL, .namespace = FALSE,
                  .all = (length(list(...)) + length(.attrs)) == 0)
xmlParent(x, ...) <- value
```

Arguments

node, value	the XML node whose state is to be modified, i.e. to which the child nodes are to be added or whose attribute list is to be changed.
..., x	This is for use in interactive settings when specifying a collection of values individuall. In programming contexts when one obtains the collection as a vector or list from another call, use the kids or .attrs parameter.
kids	when adding children to a node, this is a list of children nodes which should be of the same "type" (i.e. internal or R-level nodes) as the node argument. However, they can also be regular strings in which case they are converted to XML text nodes. For removeChildren, this is again a list which identifies the child nodes to be removed using the integer identifier of the child, or the name of the XML node (but this will only remove the first such node and not necessarily do what you expect when there are multiple nodes with the same name), or the XMLInternalNode object itself.
at	if specified, an integer identifying the position in the original list of children at which the new children should be added. The children are added after that child. This can also be a vector of indices which is as long as the number of children being added and specifies the position for each child being added. If the vector is shorter than the number of children being added, it is padded with NAs and so the corresponding children are added at the end of the list. This parameter is only implemented for internal nodes at present.
cdata	a logical value which controls whether children that are specified as strings/text are enclosed within a CDATA node when converted to actual nodes. This value is passed on to the relevant function that creates the text nodes, e.g. xmlTextNode and newXMLTextNode.
.attrs	a character vector identifying the names of the attributes. These strings can have name space prefixes, e.g. r:length and the namespaces will be resolved relative to the list supported by node to ensure those namespaces are defined.
.namespace	This is currently ignored and may never be supported. The intent is to identify on which set of attributes the operation is to perform - the name space declarations or the regular node attributes. This is a logical value indicating if TRUE that the attributes of interested are name space declarations, i.e. of the form xmlns:prefix or xmlns. If a value of FALSE is supplied this indicates that we are identifying regular attributes. Note that we can still identify attributes with a name space prefix as, e.g., ns:attr without this value
free	a logical value indicating whether to free the C-level memory associated with the child nodes that were removed. TRUE means to free that memory. This is only applicable for the internal nodes created with xmlTree and newXMLNode and related functions. It is necessary as automated garbage collection is tricky in this tree-based context spanning both R and C data structures and memory managers.
.all	a logical value indicating whether to remove all of the attributes within the XML node without having to specify them by name.
oldNode	the node which is to be replaced

newNode — the node which is to take the place of oldNode in the list of children of the parent of oldNode

suppressNamespaceWarning — a logical value or a character string. This is used to control the situation when an XML node or attribute is created with a name space prefix that currently has no definition for that node. This is not necessarily an error but can lead to one. This argument controls whether a warning is issued or if a separate function is called. A value of FALSE means not to suppress the warning and so it is issued. A value of TRUE causes the potential problem to be ignored assuming that the namespace will be added to this node or one of its ancestors at a later point. And if this value is a character string, we search for a function of that name and invoke it.

append — a logical value that indicates whether (TRUE) the specified attributes or children should be added to the existing attributes on the XML node (if any exist), or, if FALSE these should replace any existing attributes.

Value

Each of these functions returns the modified node. For an internal node, this is the same R object and only the C-level data structures have changed. For an R XMLNode object, this is is an entirely separate object from the original node. It must be inserted back into its parent "node" or context if the changes are to be seen in that wider context.

Author(s)

Duncan Temple Lang

References

libxml2 http://www.xmlsoft.org

See Also

xmlTree newXMLNode

Examples

```
b = newXMLNode("bob",
               namespace = c(r = "http://www.r-project.org",
                             omg = "https://www.omegahat.net"))

cat(saveXML(b), "\n")

addAttributes(b, a = 1, b = "xyz", "r:version" = "2.4.1", "omg:len" = 3)
cat(saveXML(b), "\n")

removeAttributes(b, "a", "r:version")
cat(saveXML(b), "\n")
```

```
removeAttributes(b, .attrs = names(xmlAttrs(b)))


addChildren(b, newXMLNode("el", "Red", "Blue", "Green",
                          attrs = c(lang ="en")))

k = lapply(letters, newXMLNode)
addChildren(b, kids = k)

cat(saveXML(b), "\n")

removeChildren(b, "a", "b", "c", "z")

  # can mix numbers and names
removeChildren(b, 2, "e")  # d and e

cat(saveXML(b), "\n")


i = xmlChildren(b)[[5]]
xmlName(i)

 # have the identifiers
removeChildren(b, kids = c("m", "n", "q"))



x <- xmlNode("a",
               xmlNode("b", "1"),
               xmlNode("c", "1"),
        "some basic text")

v = removeChildren(x, "b")

  # remove c and b
v = removeChildren(x, "c", "b")

  # remove the text and "c" leaving just b
v = removeChildren(x, 3, "c")

## Not run:
    # this won't work as the 10 gets coerced to a
    # character vector element to be combined with 'w'
    # and there is no node name 10.
 removeChildren(b, kids = c(10, "w"))

## End(Not run)


 # for R-level nodes (not internal)

z = xmlNode("arg", attrs = c(default="TRUE"),
              xmlNode("name", "foo"), xmlNode("defaultValue","1:10"))
```

```
o = addChildren(z,
                "some text",
                xmlNode("a", "a link",
                         attrs = c(href = "https://www.omegahat.net/RSXML")))
o


  # removing nodes

 doc = xmlParse("<top><a/><b/><c><d/><e>bob</e></c></top>")
 top = xmlRoot(doc)
 top

 removeNodes(list(top[[1]], top[[3]]))

    # a and c have disappeared.
 top
```

---

addNode *Add a node to a tree*

Description

This generic function allows us to add a node to a tree for different types of trees. Currently it just works for XMLHashTree, but it could be readily extended to the more general XMLFlatTree class. However, the concept in this function is to change the tree and return the node. This does not work unless the tree is directly mutable without requiring reassignment, i.e. the changes do not induce a new copy of the original tree object. DOM trees which are lists of lists of lists do not fall into this category.

Usage

```
addNode(node, parent, to, ...)
```

Arguments

node	the node to be added as a child of the parent.
parent	the parent node or identifier
to	the tree object
...	additional arguments that are understood by the different methods for the different types of trees/nodes. These can include attrs, namespace, namespaceDefinitions, .children.

Value

The new node object. For flat trees, this will be the node after it has been coerced to be compatible with a flat tree, i.e. has an id and the host tree added to it.

Author(s)

Duncan Temple Lang

References

https://www.w3.org

See Also

xmlHashTree asXMLTreeNode

Examples

```
tt = xmlHashTree()

top = addNode(xmlNode("top"), character(), tt)
addNode(xmlNode("a"), top, tt)
b = addNode(xmlNode("b"), top, tt)
c = addNode(xmlNode("c"), b, tt)
addNode(xmlNode("c"), top, tt)
addNode(xmlNode("c"), b, tt)
addNode(xmlTextNode("Some text"), c, tt)

xmlElementsByTagName(tt$top, "c")

tt
```

---

append.xmlNode *Add children to an XML node*

Description

This appends one or more XML nodes as children of an existing node.

Usage

```
append.XMLNode(to, ...)
append.xmlNode(to, ...)
```

Arguments

to	the XML node to which the sub-nodes are to be added.
...	the sub-nodes which are to be added to the to node. If this is a list of XMLNode objects (e.g. create by a call to lapply), then that list is used.

Details

append.xmlNode is a generic function with method append.XMLNode for class "XMLNode" and default method base::append.

This seems historical and users may as well use append.XMLNode directly.

Value

The original to node containing its new children nodes.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

[<-.XMLNode [[<-.XMLNode [.XMLNode [[.XMLNode

Examples

```
# Create a very simple representation of a simple dataset.
# This is just an example. The result is
 # <data numVars="2" numRecords="3">
 # <varNames>
 #  <string>
 #   A
 #  </string>
 #  <string>
 #   B
 #  </string>
 # </varNames>
 # <record>
 #  1.2 3.5
 # </record>
 # <record>
 #  20.2 13.9
 # </record>
 # <record>
 #  10.1 5.67
 # </record>
 # </data>


n = xmlNode("data", attrs = c("numVars" = 2, numRecords = 3))
n = append.xmlNode(n, xmlNode("varNames", xmlNode("string", "A"), xmlNode("string", "B")))
n = append.xmlNode(n, xmlNode("record", "1.2 3.5"))
n = append.xmlNode(n, xmlNode("record", "20.2 13.9"))
n = append.xmlNode(n, xmlNode("record", "10.1 5.67"))
```

```
 print(n)


## Not run:
   tmp <-  lapply(references, function(i) {
                                  if(!inherits(i, "XMLNode"))
                                    i <- xmlNode("reference", i)
                                  i
                                })

   r <- xmlNode("references")
   r[["references"]] <- append.xmlNode(r[["references"]], tmp)

## End(Not run)
```

---

asXMLNode *Converts non-XML node objects to XMLTextNode objects*

Description

This function is used to convert S objects that are not already XMLNode objects into objects of that class. Specifically, it treats the object as a string and creates an XMLTextNode object.

Also, there is a method for converting an XMLInternalNode - the C-level libxml representation of a node - to an explicit R-only object which contains the R values of the data in the internal node.

Usage

```
asXMLNode(x)
```

Arguments

x	the object to be converted to an XMLNode object. This is typically alread an object that inherits from XMLNode or a string.

Value

An object of class XMLNode.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlNode xmlTextNode

Examples

```
  # creates an XMLTextNode.
asXMLNode("a text node")

  # unaltered.
asXMLNode(xmlNode("p"))
```

---

asXMLTreeNode *Convert a regular XML node to one for use in a "flat" tree*

Description

This coerces a regular R-based XML node (i.e. not an internal C-level node) to a form that can be inserted into a flat tree, i.e. one that stores the nodes in a non-hierarchical manner. It is thus used in conjunction with xmlHashTree It adds id and env fields to the node and specializes the class by prefixing className to the class attribute.

This is not used very much anymore as we use the internal nodes for most purposes.

Usage

```
asXMLTreeNode(node, env, id = get(".nodeIdGenerator", env)(xmlName(node)),
              className = "XMLTreeNode")
```

Arguments

node	the original XML node
env	the XMLFlatTree object into which this node will be inserted.
id	the identifier for the node in the flat tree. If this is not specified, we consult the tree itself and its built-in identifier generator. By default, the name of the node is used as its identifier unless there is another node with that name.
className	a vector of class names to be prefixed to the existing class vector of the node.

Value

An object of class className, i.e. by default "XMLTreeNode".

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

xmlHashTree

Examples

```
txt = '<foo a="123" b="an attribute"><bar>some text</bar>other text</foo>'
doc = xmlTreeParse(txt)

class(xmlRoot(doc))

as(xmlRoot(doc), "XMLInternalNode")
```

`catalogLoad` *Manipulate XML catalog contents*

Description

These functions allow the R user to programmatically control the XML catalog table used in the XML parsing tools in the C-level libxml2 library and hence in R packages that use these, e.g. the XML and Sxslt packages. Catalogs are consulted whenever an external document needs to be loaded. XML catalogs allow one to influence how such a document is loaded by mapping document identifiers to alternative locations, for example to refer to locally available versions. They support mapping URI prefixes to local file directories/files, resolving both SYSTEM and PUBLIC identifiers used in DOCTYPE declarations at the top of an XML/HTML document, and delegating resolution to other catalog files. Catalogs are written using an XML format.

Catalogs allow resources used in XInclude nodes and XSL templates to refer to generic network URLs and have these be mapped to local files and so avoid potentially slow network retrieval. Catalog files are written in XML We might have a catalog file that contains the XML In the XDynDocs package, we refer to OmegahatXSL files and DocBook XSL files have a catalog file of the form

The functions provided here allow the R programmer to empty the current contents of the global catalog table and so start from scratch (`catalogClearTable`), load the contents of a catalog file into the global catalog table (`catalogLoad`), and to add individual entries programmatically without the need for a catalog table.

In addition to controlling the catalogs via these functions, we can use `catalogResolve` to use the catalog to resolve the name of a resource and map it to a local resource.

`catalogDump` allows us to retrieve an XML document representing the current contents of the in-memory catalog .

More information can be found at http://xmlsoft.org/catalog.html many resources and the specification for the catalog format at https://www.oasis-open.org/committees/entity/spec-2001-08-06.html.

Usage

```
catalogLoad(fileNames)
catalogClearTable()
catalogAdd(orig, replace, type = "rewriteURI")
catalogDump(fileName = tempfile(), asText = TRUE)
```

Arguments

orig	a character vector of identifiers, e.g. URIs, that are to be mapped to a different name via the catalog. This can be a named character vector where the names are the original URIs and the values are the corresponding rewritten values.
replace	a character vector of the rewritten or resolved values for the identifiers given in orig. Often this omitted and the original-rewrite pairs are given as a named vector via orig.
type	a character vector with the same length as orig (or recycled to have the same length) which specifies the type of the resources in the elements of orig. Valid values are rewriteURI, rewriteSystem, system, public.
fileNames	a character vector giving the names of the catalog files to load.
fileName	the name of the file in which to place the contents of the current catalog
asText	a logical value which indicates whether to write the catalog as a character string if filename is not specified.

Value

These functions are used for their side effects on the global catalog table maintained in C by libxml2. Their return values are logical values/vectors indicating whether the particular operation were successful or not.

References

This provides an R-like interface to a small subset of the catalog API made available in libxml2.

See Also

catalogResolve

XInclude, XSL and import/include directives.

In addition to these functions, there is an un-exported, undocumented function named catalogDump that can be used to get the contents of the (first) catalog table.

Examples

```
# Add a rewrite rule
#
#
catalogAdd(c("https://www.omegahat.net/XML" = system.file("XML", package
= "XML")))
catalogAdd("https://www.omegahat.net/XML", system.file("XML", package =
"XML"))
catalogAdd("http://www.r-project.org/doc/",
           paste(R.home(), "doc", "", sep = .Platform$file.sep))

#
#          This shows how we can load a catalog and then resolve a
#          systemidentifier that it maps.
#
```

```
catalogLoad(system.file("exampleData", "catalog.xml", package = "XML"))
catalogResolve("docbook4.4.dtd", "system")
catalogResolve("-//OASIS//DTD DocBook XML V4.4//EN", "public")
```

`catalogResolve` *Look up an element via the XML catalog mechanism*

Description

XML parsers use a catalog to map generic system and public addresses to actual local files or potentially different remote files. We can use a catalog to map a reference such as `https://www.omegahat.net/XSL/` to a particular directory on our local machine and then not have to modify any of the documents if we move the local files to another directory, e.g. install a new version in an alternate directory.

This function provides a mechanism to query the catalog to resolve a URI, PUBLIC or SYSTEM identifier.

This is now vectorized, so accepts a character vector of URIs and recycles `type` to have the same length.

If an entry is not resolved via the catalog system, a `NA` is returned for that element. To leave the value unaltered in this case, use `asIs = TRUE` .

Usage

```
catalogResolve(id, type = "uri", asIs = FALSE, debug = FALSE)
```

Arguments

`id`	the name of the (generic) element to be resolved
`type`	a string, specifying whether the lookup is for a uri, system or public element
`asIs`	a logical. If `TRUE` any element of `id` which is not resolved by the catalog system will be left as given in the call. If `FALSE`, such unresolved elements are identified by `NA`.
`debug`	logical value indicating whether to turn on debugging output written to the console (`TRUE`) or not (`FALSE`).

Value

A character vector. If the element was resolved, the single element is the resolved value. Otherwise, the character vector will contain no elements.

Author(s)

Duncan Temple Lang

References

http://www.xmlsoft.org

See Also

xmlTreeParse

Examples

```
if(!exists("Sys.setenv")) Sys.setenv = Sys.putenv

Sys.setenv("XML_CATALOG_FILES" = system.file("exampleData", "catalog.xml", package = "XML"))



catalogResolve("-//OASIS//DTD DocBook XML V4.4//EN", "public")

catalogResolve("https://www.omegahat.net/XSL/foo.xsl")

catalogResolve("https://www.omegahat.net/XSL/article.xsl", "uri")
catalogResolve("https://www.omegahat.net/XSL/math.xsl", "uri")


  # This one does not resolve anything, returning an empty value.
catalogResolve("http://www.oasis-open.org/docbook/xml/4.1.2/foo.xsl", "uri")


   # Vectorized and returns NA for the first and /tmp/html.xsl
   # for the second.

 catalogAdd("http://made.up.domain", "/tmp")
 catalogResolve(c("ddas", "http://made.up.domain/html.xsl"), asIs = TRUE)
```

---

coerceNodes *Transform between XML representations*

Description

This collection of coercion methods (i.e. `as(obj, "type")`) allows users of the `XML` package to switch between different representations of XML nodes and to map from an XML document to the root node and from a node to the document. This helps to manage the nodes

Value

An object of the target type.

See Also

xmlTreeParse xmlParse

`compareXMLDocs` *Indicate differences between two XML documents*

Description

This function is an attempt to provide some assistance in determining if two XML documents are the same and if not, how they differ. Rather than comparing the tree structure, this function compares the frequency distributions of the names of the node. It omits position, attributes, simple content from the comparison. Those are left to the functions that have more contextual information to compare two documents.

Usage

```
compareXMLDocs(a, b, ...)
```

Arguments

a, b	two parsed XML documents that must be internal documents, i.e. created with `xmlParse` or created with `newXMLNode`.
...	additional parameters that are passed on to the `summary` method for an internal document.

Value

A list with elements

`inA`	the names and counts of the XML elements that only appear in the first document
`inB`	the names and counts of the XML elements that only appear in the second document
`countDiffs`	a vector giving the difference in number of nodes with a particular name.

These give a description of what is missing from one document relative to the other.

Author(s)

Duncan Temple Lang

See Also

`getNodeSet`

Examples

```
tt =
 '<x>
    <a>text</a>
    <b foo="1"/>
    <c bar="me">
       <d>a phrase</d>
    </c>
  </x>'

  a = xmlParse(tt, asText = TRUE)
  b = xmlParse(tt, asText = TRUE)
  d = getNodeSet(b, "//d")[[1]]
  xmlName(d) = "bob"
  addSibling(xmlParent(d), newXMLNode("c"))

  compareXMLDocs(a, b)
```

---

docName *Accessors for name of XML document*

Description

These functions and methods allow us to query and set the "name" of an XML document. This is intended to be its URL or file name or a description of its origin if raw XML content provided as a string.

Usage

```
docName(doc, ...)
```

Arguments

doc	the XML document object, of class `XMLInternalDocument` or `XMLDocument`.
...	additional methods for methods

Value

A character string giving the name. If the document was created from text, this is `NA` (of class character).

The assignment function returns the updated object, but the R assignment operation will return the value on the right of the assignment!

Author(s)

Duncan Temple Lang

See Also

xmlTreeParse xmlInternalTreeParse newXMLDoc

Examples

```
f = system.file("exampleData", "catalog.xml",  package = "XML")
doc = xmlInternalTreeParse(f)
docName(doc)

doc = xmlInternalTreeParse("<a><b/></a>", asText = TRUE)
    # an NA
docName(doc)
docName(doc) = "Simple XML example"
docName(doc)
```

Doctype *Constructor for DTD reference*

Description

This is a constructor for the Doctype class that can be provided at the top of an XML document to provide information about the class of document, i.e. its DTD or schema. Also, there is a method for converting such a Doctype object to a character string.

Usage

```
Doctype(system = character(), public = character(), name = "")
```

Arguments

system	the system URI that locates the DTD.
public	the identifier for locating the DTD in a catalog, for example. This should be a character vector of length 2, giving the public identifier and a URI. If just the public identifier is given and a string is given for system argument, the system value is used as the second element of public. The public identifer should be of the form +//creator//name//language where the first element is either + or -, and the language is described by a code in the ISO 639 document.
name	the name of the root element in the document. This should be the first parameter, but is left this way for backward compatability. And

Value

An object of class Doctype.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/ XML Elements of Style, Simon St. Laurent.

See Also

saveXML

Examples

```
d = Doctype(name = "section",
            public = c("-//OASIS//DTD DocBook XML V4.2//EN",
                       "http://oasis-open.org/docbook/xml/4.2/docbookx.dtd"))
as(d, "character")

 # this call switches the system to the URI associated with the PUBLIC element.
d = Doctype(name = "section",
            public = c("-//OASIS//DTD DocBook XML V4.2//EN"),
            system = "http://oasis-open.org/docbook/xml/4.2/docbookx.dtd")
```

Doctype-class *Class to describe a reference to an XML DTD*

Description

This class is intended to identify a DTD by SYSTEM file and/or PUBLIC catalog identifier. This is used in the DOCTYPE element of an XML document.

Objects from the Class

Objects can be created by calls to the constructor function Doctype.

Slots

`name`: Object of class `"character"`. This is the name of the top-level element in the XML document.

`system`: Object of class `"character"`. This is the name of the file on the system where the DTD document can be found. Can this be a URI?

`public`: Object of class `"character"`. This gives the PUBLIC identifier for the DTD that can be searched for in a catalog, for example to map the DTD reference to a local system element.

Methods

There is a constructor function and also methods for coerce to convert an object of this class to a character.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.xmlsoft.org

See Also

Doctype saveXML

Examples

```
d = Doctype(name = "section",
            public = c("-//OASIS//DTD DocBook XML V4.2//EN",
                       "http://oasis-open.org/docbook/xml/4.2/docbookx.dtd"))
```

---

ensureNamespace *Ensure that the node has a definition for particular XML namespaces*

Description

This function is a helper function for use in creating XML content. We often want to create a node that will be part of a larger XML tree and use a particular namespace for that node name. Rather than defining the namespace in each new node, we want to ensure that it is define on an ancestor node. This function aids in that task. We call the function with the ancestor node or top-level document and have it check whether the namespace is already defined or have it add it to the node and return.

This is intended for use with XMLInternalNode objects which are direclty mutable (rather than changing a copy of the node and having to insert that back into the larger tree.)

Usage

```
ensureNamespace(doc, what)
```

Arguments

doc	an XMLInternalDocument or XMLInternalNode on which the namespace is to be defined. If this is a documentm, we use the root node.
what	a named character vector giving the URIs for the namespace definitions and the names giving the desired prefixes

Value

This is used for the potential side effects of modifying the XML node to add (some of) the namespaces as needed.

Author(s)

Duncan Temple Lang

References

XML namespaces

See Also

newXMLNamespace newXMLNode

Examples

```
doc = newXMLDoc()
top = newXMLNode("article", doc = doc)
ensureNamespace(top, c(r = "http://www.r-project.org"))
b = newXMLNode("r:code", parent = top)
print(doc)
```

---

findXInclude *Find the XInclude node associated with an XML node*

Description

This function is used to traverse the ancestors of an internal XML node to find the associated XInclude node that identifies it as being an XInclude'd node. Each top-level node that results from an include href=... in the libxml2 parser is sandwiched between nodes of class XMLXIncludeStartNode and XMLXIncludeStartNode. These are the sibling nodes.

Another approach to finding the origin of the XInclude for a given node is to search for an attribute xml:base. This only works if the document being XInclude'd is in a different directory than the base document. If this is the case, we can use an XPath query to find the node containing the attribute via "./ancestor::*[@xml:base]".

Usage

```
findXInclude(x, asNode = FALSE, recursive = FALSE)
```

Arguments

x	the node whose XInclude "ancestor" is to be found
asNode	a logical value indicating whether to return the node itself or the attributes of the node which are typically the immediately interesting aspect of the node.
recursive	a logical value that controls whether the full path of the nested includes is returned or just the path in the immediate XInclude element.

Value

Either NULL if there was no node of class XMLXIncludeStartNode found. Otherwise, if asNode is TRUE, that XMLXIncludeStartNode node is returned, or alternatively its attribute character vector.

Author(s)

Duncan Temple Lang

References

www.libxml.org

See Also

xmlParse and the xinclude parameter.

Examples

```
f = system.file("exampleData", "functionTemplate.xml", package = "XML")

cat(readLines(f), "\n")

doc = xmlParse(f)

 # Get all the para nodes
 # We just want to look at the 2nd and 3rd which are repeats of the
 # first one.
a = getNodeSet(doc, "//author")
findXInclude(a[[1]])

i = findXInclude(a[[1]], TRUE)
top = getSibling(i)

  # Determine the top-level included nodes
tmp = getSibling(i)
nodes = list()
while(!inherits(tmp, "XMLXIncludeEndNode")) {
  nodes = c(nodes, tmp)
  tmp = getSibling(tmp)
}
```

---

free *Release the specified object and clean up its memory usage*

Description

This generic function is available for explicitly releasing the memory associated with the given object. It is intended for use on external pointer objects which do not have an automatic finalizer function/routine that cleans up the memory that is used by the native object. This is the case, for example, for an XMLInternalDocument. We cannot free it with a finalizer in all cases as we may have a reference to a node in the associated document tree. So the user must explicitly release the XMLInternalDocument object to free the memory it occupies.

Usage

```
free(obj)
```

Arguments

obj	the object whose memory is to be released, typically an external pointer object or object that contains a slot that is an external pointer.

Details

The methods will generally call a C routine to free the native memory.

Value

An updated version of the object with the external address set to NIL. This is up to the individual methods.

Author(s)

Duncan Temple Lang

See Also

xmlTreeParse with useInternalNodes = TRUE

Examples

```
f = system.file("exampleData", "boxplot.svg", package = "XML")
doc = xmlParse(f)
nodes = getNodeSet(doc, "//path")
rm(nodes)
# free(doc)
```

---

genericSAXHandlers *SAX generic callback handler list*

Description

This is a convenience function to get the collection of generic functions that make up the callbacks for the SAX parser. The return value can be used directly as the value of the handlers argument in xmlEventParse. One can easily specify a subset of the handlers by giving the names of the elements to include or exclude.

Usage

```
genericSAXHandlers(include, exclude, useDotNames = FALSE)
```

Arguments

include	if supplied, this gives the names of the subset of elements to return.
exclude	if supplied (and include is not), this gives the names of the elements to remove from the list of functions.
useDotNames	a logical value. If this is TRUE, the names of the elements in the list of handler functions are prefixed with '.'. This is the newer format used to differentiate general element handlers and node-name-specific handlers.

Value

A list of functions. By default, the elements are named startElement, endElement, comment, text, processingInstruction, entityDeclaration and contain the corresponding generic SAX callback function, i.e. given by the element name with the .SAX suffix.

If include or exclude is specified, a subset of this list is returned.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlEventParse startElement.SAX endElement.SAX comment.SAX processingInstruction.SAX entityDeclaration.SAX .InitSAXMethods

Examples

---

getChildrenStrings *Get the individual*

Description

This is different from xmlValue applied to the node. That concatenates all of the text in the child nodes (and their descendants) This is a faster version of xmlSApply(node, xmlValue)

Usage

```
getChildrenStrings(node, encoding = getEncoding(node),
                   asVector = TRUE, len = xmlSize(node), addNames = TRUE)
```

Arguments

node	the parent node whose child nodes we want to process
encoding	the encoding to use for the text. This should come from the document itself. However, it can be useful to specify it if the encoding has not been set for the document (e.g. if we are constructing it node-by-node).
asVector	a logical value that controls whether the result is returned as a character vector or as a list (FALSE).
len	an integer giving the number of elements we expect returned. This is best left unspecified but can be provided if the caller already knows the number of child nodes. This avoids recomputing this and so provides a marginal speedup.
addNames	a logical value that controls whether we add the element names to each element of the resulting vector. This makes it easier to identify from which element each string came.

Value

A character vector.

Author(s)

Duncan Temple Lang

See Also

xmlValue

Examples

```
doc = xmlParse("<doc><a>a string</a> some text <b>another</b></doc>")
getChildrenStrings(xmlRoot(doc))

doc = xmlParse("<doc><a>a string</a> some text <b>another</b><c/><d>abc<e>xyz</e></d></doc>")
getChildrenStrings(xmlRoot(doc))
```

---

getEncoding *Determines the encoding for an XML document or node*

Description

This function and its methods are intended to return the encoding of n XML . It is similar to Encoding but currently restricted to XML nodes and documents.

Usage

```
getEncoding(obj, ...)
```

Arguments

obj	the object whose encoding is being queried.
...	any additional parameters which can be customized by the methods.

Value

A character vector of length 1 giving the encoding of the XML document.

Author(s)

Duncan Temple Lang

Examples

```
f = system.file("exampleData", "charts.svg", package = "XML")
doc = xmlParse(f)
getEncoding(doc)
n = getNodeSet(doc, "//g/text")[[1]]
getEncoding(n)

f = system.file("exampleData", "iTunes.plist", package = "XML")
doc = xmlParse(f)
getEncoding(doc)
```

---

getHTMLLinks *Get links or names of external files in HTML document*

Description

These functions allow us to retrieve either the links within an HTML document, or the collection of names of external files referenced in an HTML document. The external files include images, JavaScript and CSS documents.

Usage

```
getHTMLLinks(doc, externalOnly = TRUE, xpQuery = "//a/@href",
               baseURL = docName(doc), relative = FALSE)
getHTMLExternalFiles(doc, xpQuery = c("//img/@src", "//link/@href",
                                      "//script/@href", "//embed/@src"),
                     baseURL = docName(doc), relative = FALSE,
                     asNodes = FALSE, recursive = FALSE)
```

Arguments

doc	the HTML document as a URL, local file name, parsed document or an XML/HTML node
externalOnly	a logical value that indicates whether we should only return links to external documents and not references to internal anchors/nodes within this document, i.e. those that of the form #foo.
xpQuery	a vector of XPath elements which match the elements of interest
baseURL	the URL of the container document. This is used to resolve relative references/links.
relative	a logical value indicating whether to leave the references as relative to the base URL or to expand them to their full paths.
asNodes	a logical value that indicates whether we want the actual HTML/XML nodes in the document that reference external documents or just the names of the external documents.
recursive	a logical value that controls whether we recursively process the external documents we find in the top-level document examining them for their external files.

Value

getHTMLLinks returns a character vector of the links.

getHTMLExternalFiles returns a character vector.

Author(s)

Duncan Temple Lang

See Also

getXIncludes

Examples

```
## Not run:  # domain lost
  try(getHTMLLinks("https://www.omegahat.net"))

  try(getHTMLLinks("https://www.omegahat.net/RSXML"))

  try(unique(getHTMLExternalFiles("https://www.omegahat.net")))

## End(Not run)
```

getLineNumber *Determine the location - file & line number of an (internal) XML node*

Description

The getLineNumber function is used to query the location of an internal/C-level XML node within its original "file". This gives us the line number. getNodeLocation gives both the line number and the name of the file in which the node is located, handling XInclude files in a top-level document and identifying the included file, as appropriate. getNodePosition returns a simplified version of getNodeLocation, combining the file and line number into a string and ignoring the XPointer component.

This is useful when we identify a node with a particular charactestic and want to view/edit the original document, e.g. when authoring an Docbook article.

Usage

```
getLineNumber(node, ...)
getNodeLocation(node, recursive = TRUE, fileOnly = FALSE)
getNodePosition(x)
```

Arguments

node, x	the node whose location or line number is of interest
...	additional parameters for methods should they be defined.
recursive	a logical value that controls whether the full path of the nested includes is returned or just the path in the immediate XInclude element.
fileOnly	a logical value which if TRUE means that only the name of the file is returned, and not the xpointer attribute or line number .

Value

getLineNumber returns an integer. getNodeLocation returns a list with two elements - file and line which are a character string and the integer line number.

For text nodes, the line number is taken from the previous sibling nodes or the parent node.

Author(s)

Duncan Temple Lang

References

libxml2

See Also

findXInclude xmlParse getNodeSet xpathApply

Examples

```
f = system.file("exampleData", "xysize.svg", package = "XML")
doc = xmlParse(f)
e = getNodeSet(doc, "//ellipse")
sapply(e, getLineNumber)
```

`getNodeSet` *Find matching nodes in an internal XML tree/DOM*

Description

These functions provide a way to find XML nodes that match a particular criterion. It uses the XPath syntax and allows very powerful expressions to identify nodes of interest within a document both clearly and efficiently. The XPath language requires some knowledge, but tutorials are available on the Web and in books. XPath queries can result in different types of values such as numbers, strings, and node sets. It allows simple identification of nodes by name, by path (i.e. hierarchies or sequences of node-child-child...), with a particular attribute or matching a particular attribute with a given value. It also supports functionality for navigating nodes in the tree within a query (e.g. `ancestor()`, `child()`, `self()`), and also for manipulating the content of one or more nodes (e.g. `text`). And it allows for criteria identifying nodes by position, etc. using some counting operations. Combining XPath with R allows for quite flexible node identification and manipulation. XPath offers an alternative way to find nodes of interest than recursively or iteratively navigating the entire tree in R and performing the navigation explicitly.

One can search an entire document or start the search from a particular node. Such node-based searches can even search up the tree as well as within the sub-tree that the node parents. Node specific XPath expressions are typically started with a "." to indicate the search is relative to that node.

You can use several XPath 2.0 functions in the XPath query. Furthermore, you can also register additional XPath functions that are implemented either with R functions or C routines. (See `xpathFuns`.)

The set of matching nodes corresponding to an XPath expression are returned in R as a list. One can then iterate over these elements to process the nodes in whatever way one wants. Unfortunately, this involves two loops - one in the XPath query over the entire tree, and another in R. Typically, this is fine as the number of matching nodes is reasonably small. However, if repeating this on numerous files, speed may become an issue. We can avoid the second loop (i.e. the one in R) by applying a function to each node before it is returned to R as part of the node set. The result of the function call is then returned, rather than the node itself.

One can provide an R expression rather than an R function for `fun`. This is expected to be a call and the first argument of the call will be replaced with the node.

Dealing with expressions that relate to the default namespaces in the XML document can be confusing.

`xpathSApply` is a version of `xpathApply` which attempts to simplify the result if it can be converted to a vector or matrix rather than left as a list. In this way, it has the same relationship to `xpathApply` as `sapply` has to `lapply`.

matchNamespaces is a separate function that is used to facilitate specifying the mappings from namespace prefix used in the XPath expression and their definitions, i.e. URIs, and connecting these with the namespace definitions in the target XML document in which the XPath expression will be evaluated.

matchNamespaces uses rules that are very slightly awkard or specifically involve a special case. This is because this mapping of namespaces from XPath to XML targets is difficult, involving prefixes in the XPath expression, definitions in the XPath evaluation context and matches of URIs with those in the XML document. The function aims to avoid having to specify all the prefix=uri pairs by using "sensible" defaults and also matching the prefixes in the XPath expression to the corresponding definitions in the XML document.

The rules are as follows. namespaces is a character vector. Any element that has a non-trivial name (i.e. other than "") is left as is and the name and value define the prefix = uri mapping. Any elements that have a trivial name (i.e. no name at all or "") are resolved by first matching the prefix to those of the defined namespaces anywhere within the target document, i.e. in any node and not just the root one. If there is no match for the first element of the namespaces vector, this is treated specially and is mapped to the default namespace of the target document. If there is no default namespace defined, an error occurs.

It is best to give explicit the argument in the form c(prefix = uri, prefix = uri). However, one can use the same namespace prefixes as in the document if one wants. And one can use an arbitrary namespace prefix for the default namespace URI of the target document provided it is the first element of namespaces.

See the 'Details' section below for some more information.

Usage

```
getNodeSet(doc, path, namespaces = xmlNamespaceDefinitions(doc, simplify = TRUE),
                 fun = NULL, sessionEncoding = CE_NATIVE, addFinalizer = NA, ...)
xpathApply(doc, path, fun, ... ,
            namespaces =  xmlNamespaceDefinitions(doc, simplify = TRUE),
              resolveNamespaces = TRUE, addFinalizer = NA, xpathFuns = list())
xpathSApply(doc, path, fun = NULL, ... ,
             namespaces = xmlNamespaceDefinitions(doc, simplify = TRUE),
               resolveNamespaces = TRUE, simplify = TRUE,
                addFinalizer = NA)
matchNamespaces(doc, namespaces,
           nsDefs = xmlNamespaceDefinitions(doc, recursive = TRUE, simplify = FALSE),
                defaultNs = getDefaultNamespace(doc, simplify = TRUE))
```

Arguments

doc	an object of class XMLInternalDocument
path	a string (character vector of length 1) giving the XPath expression to evaluate.
namespaces	a named character vector giving the namespace prefix and URI pairs that are to be used in the XPath expression and matching of nodes. The prefix is just a simple string that acts as a short-hand or alias for the URI that is the unique identifier for the namespace. The URI is the element in this vector and the prefix is the corresponding element name. One only needs to specify the namespaces

in the XPath expression and for the nodes of interest rather than requiring all the namespaces for the entire document. Also note that the prefix used in this vector is local only to the path. It does not have to be the same as the prefix used in the document to identify the namespace. However, the URI in this argument must be identical to the target namespace URI in the document. It is the namespace URIs that are matched (exactly) to find correspondence. The prefixes are used only to refer to that URI.

`fun` a function object, or an expression or call, which is used when the result is a node set and evaluated for each node element in the node set. If this is a call, the first argument is replaced with the current node.

`...` any additional arguments to be passed to `fun` for each node in the node set.

`resolveNamespaces`
a logical value indicating whether to process the collection of namespaces and resolve those that have no name by looking in the default namespace and the namespace definitions within the target document to match by prefix.

`nsDefs` a list giving the namespace definitions in which to match any prefixes. This is typically computed directly from the target document and the default value is most appropriate.

`defaultNs` the default namespace prefix-URI mapping given as a named character vector. This is not a namespace definition object. This is used when matching a simple prefix that has no corresponding entry in `nsDefs` and is the first element in the `namespaces` vector.

`simplify` a logical value indicating whether the function should attempt to perform the simplification of the result into a vector rather than leaving it as a list. This is the same as `sapply` does in comparison to `lapply`.

`sessionEncoding`
experimental functionality and parameter related to encoding.

`addFinalizer` a logical value or identifier for a C routine that controls whether we register finalizers on the intenal node.

`xpathFuns` a list containing either character strings, functions or named elements containing the address of a C routine. These identify functions that can be used in the XPath expression. A character string identifies the name of the XPath function and the R function of the same name (and located on the R search path). A C routine to implement an XPath function is specified via a call to `getNativeSymbolInfo` and passing just the address field. This is provided in the `list()` with a name which is used as the name of the XPath function.

Details

When a namespace is defined on a node in the XML document, an XPath expressions must use a namespace, even if it is the default namespace for the XML document/node. For example, suppose we have an XML document `<help xmlns="http://www.r-project.org/Rd"><topic>...</topic></help>` To find all the topic nodes, we might want to use the XPath expression `"/help/topic"`. However, we must use an explicit namespace prefix that is associated with the URI `http://www.r-project.org/Rd` corresponding to the one in the XML document. So we would use `getNodeSet(doc, "/r:help/r:topic", c(r = "http://www.r-project.org/Rd"))`.

As described above, the functions attempt to allow the namespaces to be specified easily by the R user and matched to the namespace definitions in the target document.

This calls the libxml routine `xmlXPathEval`.

Value

The results can currently be different based on the returned value from the XPath expression evaluation:

`list`	a node set
`numeric`	a number
`logical`	a boolean
`character`	a string, i.e. a single character element.

If `fun` is supplied and the result of the XPath query is a node set, the result in R is a list.

Note

In order to match nodes in the default name space for documents with a non-trivial default namespace, e.g. given as `xmlns="https://www.omegahat.net"`, you will need to use a prefix for the default namespace in this call. When specifying the namespaces, give a name - any name - to the default namespace URI and then use this as the prefix in the XPath expression, e.g. `getNodeSet(d, "//d:myNode", c(d = "https://www.omegahat.net"))` to match myNode in the default name space `https://www.omegahat.net`.

This default namespace of the document is now computed for us and is the default value for the namespaces argument. It can be referenced using the prefix 'd', standing for default but sufficiently short to be easily used within the XPath expression.

More of the XPath functionality provided by libxml can and may be made available to the R package. Facilities such as compiled XPath expressions, functions, ordered node information are examples.

Please send requests to the package maintainer.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://xmlsoft.org, https://www.w3.org/XML/, https://www.w3.org/TR/xpath/

See Also

`xmlTreeParse` with `useInternalNodes` as `TRUE`.

Examples

```
doc = xmlParse(system.file("exampleData", "tagnames.xml", package = "XML"))

els = getNodeSet(doc, "/doc//a[@status]")
sapply(els, function(el) xmlGetAttr(el, "status"))

  # use of namespaces on an attribute.
getNodeSet(doc, "/doc//b[@x:status]", c(x = "https://www.omegahat.net"))
getNodeSet(doc, "/doc//b[@x:status='foo']", c(x = "https://www.omegahat.net"))

  # Because we know the namespace definitions are on /doc/a
  # we can compute them directly and use them.
nsDefs = xmlNamespaceDefinitions(getNodeSet(doc, "/doc/a")[[1]])
ns = structure(sapply(nsDefs, function(x) x$uri), names = names(nsDefs))
getNodeSet(doc, "/doc//b[@omegahat:status='foo']", ns)[[1]]

# free(doc)

#####
f = system.file("exampleData", "eurofxref-hist.xml.gz", package = "XML")
e = xmlParse(f)
ans = getNodeSet(e, "//o:Cube[@currency='USD']", "o")
sapply(ans, xmlGetAttr, "rate")

 # or equivalently
ans = xpathApply(e, "//o:Cube[@currency='USD']", xmlGetAttr, "rate", namespaces = "o")
# free(e)



 # Using a namespace
f = system.file("exampleData", "SOAPNamespaces.xml", package = "XML")
z = xmlParse(f)
getNodeSet(z, "/a:Envelope/a:Body", c("a" = "http://schemas.xmlsoap.org/soap/envelope/"))
getNodeSet(z, "//a:Body", c("a" = "http://schemas.xmlsoap.org/soap/envelope/"))
# free(z)


 # Get two items back with namespaces
f = system.file("exampleData", "gnumeric.xml", package = "XML")
z = xmlParse(f)
getNodeSet(z, "//gmr:Item/gmr:name", c(gmr="http://www.gnome.org/gnumeric/v2"))

#free(z)

#####
# European Central Bank (ECB) exchange rate data

 # Data is available from "http://www.ecb.int/stats/eurofxref/eurofxref-hist.xml"
 # or locally.

uri = system.file("exampleData", "eurofxref-hist.xml.gz", package = "XML")
```

```
doc = xmlParse(uri)

  # The default namespace for all elements is given by
namespaces <- c(ns="http://www.ecb.int/vocabulary/2002-08-01/eurofxref")


    # Get the data for Slovenian currency for all time periods.
    # Find all the nodes of the form <Cube currency="SIT"...>

slovenia = getNodeSet(doc, "//ns:Cube[@currency='SIT']", namespaces )

   # Now we have a list of such nodes, loop over them
   # and get the rate attribute
rates = as.numeric( sapply(slovenia, xmlGetAttr, "rate") )
   # Now put the date on each element
   # find nodes of the form <Cube time=".." ... >
   # and extract the time attribute
names(rates) = sapply(getNodeSet(doc, "//ns:Cube[@time]", namespaces ),
                      xmlGetAttr, "time")

   #  Or we could turn these into dates with strptime()
strptime(names(rates), "%Y-%m-%d")


  #  Using xpathApply, we can do
rates = xpathApply(doc, "//ns:Cube[@currency='SIT']",
                   xmlGetAttr, "rate", namespaces = namespaces )
rates = as.numeric(unlist(rates))

  # Using an expression rather than  a function and ...
rates = xpathApply(doc, "//ns:Cube[@currency='SIT']",
                   quote(xmlGetAttr(x, "rate")), namespaces = namespaces )

#free(doc)

  #
 uri = system.file("exampleData", "namespaces.xml", package = "XML")
 d = xmlParse(uri)
 getNodeSet(d, "//c:c", c(c="http://www.c.org"))

getNodeSet(d, "/o:a//c:c", c("o" = "https://www.omegahat.net", "c" = "http://www.c.org"))

  # since https://www.omegahat.net is the default namespace, we can
  # just the prefix "o" to map to that.
 getNodeSet(d, "/o:a//c:c", c("o", "c" = "http://www.c.org"))


  # the following, perhaps unexpectedly but correctly, returns an empty
  # with no matches

 getNodeSet(d, "//defaultNs", "https://www.omegahat.net")

  # But if we create our own prefix for the evaluation of the XPath
```

```
 # expression and use this in the expression, things work as one
 # might hope.
getNodeSet(d, "//dummy:defaultNs", c(dummy = "https://www.omegahat.net"))

 # And since the default value for the namespaces argument is the
 # default namespace of the document, we can refer to it with our own
 # prefix given as
getNodeSet(d, "//d:defaultNs", "d")

 # And the syntactic sugar is
d["//d:defaultNs", namespace = "d"]


 # this illustrates how we can use the prefixes in the XML document
 # in our query and let getNodeSet() and friends map them to the
 # actual namespace definitions.
 # "o" is used to represent the default namespace for the document
 # i.e. https://www.omegahat.net, and "r" is mapped to the same
 # definition that has the prefix "r" in the XML document.

tmp = getNodeSet(d, "/o:a/r:b/o:defaultNs", c("o", "r"))
xmlName(tmp[[1]])


#free(d)


 # Work with the nodes and their content (not just attributes) from the node set.
 # From bondsTables.R in examples/

## Not run: ## fails to download as from May 2017
  doc =
 htmlTreeParse("http://finance.yahoo.com/bonds/composite_bond_rates?bypass=true",
               useInternalNodes = TRUE)
  if(is.null(xmlRoot(doc)))
     doc = htmlTreeParse("http://finance.yahoo.com/bonds?bypass=true",
    useInternalNodes = TRUE)

     # Use XPath expression to find the nodes
     #  <div><table class="yfirttbl">..
     # as these are the ones we want.

  if(!is.null(xmlRoot(doc))) {

   o = getNodeSet(doc, "//div/table[@class='yfirttbl']")
}

    # Write a function that will extract the information out of a given table node.
   readHTMLTable =
   function(tb)
    {
          # get the header information.
      colNames = sapply(tb[["thead"]][["tr"]]["th"], xmlValue)
```

```
      vals = sapply(tb[["tbody"]]["tr"],  function(x) sapply(x["td"], xmlValue))
      matrix(as.numeric(vals[-1,]),
             nrow = ncol(vals),
             dimnames = list(vals[1,], colNames[-1]),
             byrow = TRUE
            )
    }


     # Now process each of the table nodes in the o list.
    tables = lapply(o, readHTMLTable)
    names(tables) = lapply(o, function(x) xmlValue(x[["caption"]]))

## End(Not run)


     # this illustrates an approach to doing queries on a sub tree
     # within the document.
     # Note that there is a memory leak incurred here as we create a new
     # XMLInternalDocument in the getNodeSet().

    f = system.file("exampleData", "book.xml", package = "XML")
    doc = xmlParse(f)
    ch = getNodeSet(doc, "//chapter")
    xpathApply(ch[[2]], "//section/title", xmlValue)

      # To fix the memory leak, we explicitly create a new document for
      # the subtree, perform the query and then free it _when_ we are done
      # with the resulting nodes.
    subDoc = xmlDoc(ch[[2]])
    xpathApply(subDoc, "//section/title", xmlValue)
    free(subDoc)


    txt =
'<top xmlns="http://www.r-project.org" xmlns:r="http://www.r-project.org"><r:a><b/></r:a></top>'
    doc = xmlInternalTreeParse(txt, asText = TRUE)

## Not run:
     # Will fail because it doesn't know what the namespace x is
     # and we have to have one eventhough it has no prefix in the document.
    xpathApply(doc, "//x:b")

## End(Not run)
      # So this is how we do it - just  say x is to be mapped to the
      # default unprefixed namespace which we shall call x!
    xpathApply(doc, "//x:b", namespaces = "x")

       # Here r is mapped to the the corresponding definition in the document.
    xpathApply(doc, "//r:a", namespaces = "r")
       # Here, xpathApply figures this out for us, but will raise a warning.
    xpathApply(doc, "//r:a")
```

```
   # And here we use our own binding.
xpathApply(doc, "//x:a", namespaces = c(x = "http://www.r-project.org"))



   # Get all the nodes in the entire tree.
table(unlist(sapply(doc["//*|//text()|//comment()|//processing-instruction()"],
class)))



 ## Use of XPath 2.0 functions min() and max()
 doc = xmlParse('<doc><p age="10"/><p age="12"/><p age="7"/></doc>')
 getNodeSet(doc, "//p[@age  = min(//p/@age)]")
 getNodeSet(doc, "//p[@age  = max(//p/@age)]")

 avg = function(...) {
        mean(as.numeric(unlist(...)))
       }
 getNodeSet(doc, "//p[@age > avg(//p/@age)]", xpathFuns = "avg")


doc = xmlParse('<doc><ev date="2010-12-10"/><ev date="2011-3-12"/><ev date="2015-10-4"/></doc>')
 getNodeSet(doc, "//ev[month-from-date(@date) > 7]",
            xpathFuns = list("month-from-date" =
                             function(node) {
                           match(months(as.Date(as.character(node[[1]]))), month.name)
                             }))
```

---

getRelativeURL *Compute name of URL relative to a base URL*

Description

This function is a convenience function for computing the fullly qualified URI of a document relative to a base URL. It handles the case where the document is already fully qualified and so ignores the base URL or, alternatively, is a relative document name and so prepends the base URL. It does not (yet) try to be clever by collapsing relative directories such as "..".

Usage

```
getRelativeURL(u, baseURL, sep = "/", addBase = TRUE,
               simplify = TRUE, escapeQuery = FALSE)
```

Arguments

u	the location of the target document whose fully qualified URI is to be determined.
baseURL	the base URL relative to which the value of u should be interpreted.

sep	the separator to use to separate elements of the path. For external URLs (e.g. accessed via HTTP, HTTPS, FTP), / should be used. For local files on Windows machines one might use .Platform$file.sep, but this is incorrect unless one knows that the resulting file is to be accessed using Windows file system notation, i.e. C:\\my\\folder\\file.
addBase	a logical controlling whether we prepend the base URL to the result.
simplify	a logical value that controls whether we attempt to simplify/normalize the path to remove .. and .
escapeQuery	a logical value. Currently ignored.

Details

This uses the function parseURI to compute the components of the different URIs.

Value

A character string giving the fully qualified URI for u.

Author(s)

Duncan Temple Lang

See Also

parseURI which uses the libxml2 facilities for parsing URIs.

xmlParse, xmlTreeParse, xmlInternalTreeParse. XInclude and XML Schema import/include elements for computing relative locations of included/imported files..

Examples

```
getRelativeURL("https://www.omegahat.net", "http://www.r-project.org")

getRelativeURL("bar.html", "http://www.r-project.org/")

getRelativeURL("../bar.html", "http://www.r-project.org/")
```

---

getSibling *Manipulate sibling XML nodes*

Description

These functions allow us to both access the sibling node to the left or right of a given node and so walk the chain of siblings, and also to insert a new sibling

Usage

```
getSibling(node, after = TRUE, ...)
addSibling(node, ..., kids = list(...), after = NA)
```

Arguments

node	the internal XML node (XMLInternalNode) whose siblings are of interest
...	the XML nodes to add as siblings or children to node.
kids	a list containing the XML nodes to add as siblings. This is equivalent to ... but used when we already have the nodes in a list rather than as individual objects. This is used in programmatic calls to addSibling rather interactive use where we more commonly have the individual node objects.
after	a logical value indicating whether to retrieve or add the nodes to the right (TRUE) or to the left (FALSE) of this sibling.

Value

getSibling returns an object of class XMLInternalNode (or some derived S3 class, e.g. XMLInternalTextNode)

addSibling returns a list whose elements are the newly added XML (internal) nodes.

See Also

xmlChildren, addChildren removeNodes replaceNodes

Examples

```
      # Reading Apple's iTunes files
  #
 #        Here we read  a "censored" "database" of songs from Apple's  iTune application
  #           which is stored in a property list.  The format is quite generic and
  #            the fields for each song are given in the form
  #
  #             <key>Artist</key><string>Person's name</string>
  #
  #           So to find the names of the artists for all the songs, we want to
  #           find all the <key>Artist<key> nodes and then get their next sibling
  #           which has the actual value.
  #
  #           More information can be found in .
  #
       fileName = system.file("exampleData", "iTunes.plist", package = "XML")

       doc = xmlParse(fileName)
       nodes = getNodeSet(doc, "//key[text() = 'Artist']")
       sapply(nodes, function(x)  xmlValue(getSibling(x)))


     f = system.file("exampleData", "simple.xml", package = "XML")
     tt = as(xmlParse(f), "XMLHashTree")

      tt

     e = getSibling(xmlRoot(tt)[[1]])
       # and back to the first one again by going backwards along the sibling list.
```

```
getSibling(e, after = FALSE)


   # This also works for multiple top-level "root" nodes
f = system.file("exampleData", "job.xml", package = "XML")
tt = as(xmlParse(f), "XMLHashTree")
 x = xmlRoot(tt, skip = FALSE)
 getSibling(x)
 getSibling(getSibling(x), after = FALSE)
```

---

getXIncludes *Find the documents that are XInclude'd in an XML document*

Description

The `getXMLIncludes` function finds the names of the documents that are XIncluded in a given XML document, optionally processing these documents recursively.

`xmlXIncludes` returns the hierarchy of included documents.

Usage

```
getXIncludes(filename, recursive = TRUE, skip = character(),
             omitPattern = "\\.(js|html?|txt|R|c)$",
             namespace = c(xi = "https://www.w3.org/2003/XInclude"),
            duplicated = TRUE)
xmlXIncludes(filename, recursive = TRUE,
         omitPattern = "\\.(js|html?|txt|R|c)$",
         namespace = c(xi = "https://www.w3.org/2003/XInclude"),
         addNames = TRUE,
         clean = NULL, ignoreTextParse = FALSE)
```

Arguments

`filename`	the name of the XML document's URL or file or the parsed document itself.
`recursive`	a logical value controlling whether to recursively process the XInclude'd files for their XInclude'd files
`skip`	a character vector of file names to ignore or skip over
`omitPattern`	a regular expression for indentifying files that are included that we do not want to recursively process
`namespace`	the namespace to use for the XInclude. There are two that are in use 2001 and 2003.
`duplicated`	a logical value that controls whether only the unique names of the files are returned, or if we get all references to all files.
`addNames`	a logical that controls whether we add the name of the parent file as the names vector for the collection of included file names. This is useful, but sometimes we want to disable this, e.g. to create a `JSON` representation of the hierarchy for use in, e.g., D3.

clean — how to process the names of the files. This can be a function or a character vector of two regular expressions passed to gsub. The function is called with a vector of file names. The regular expressions are used in a call to gsub.

ignoreTextParse — if TRUE, ignore the XIncluded files that are identified as text and not XML with parse="text".

Value

If recursive is FALSE, a character vector giving the names of the included files.

For recursive is TRUE, currently the same character vector form. However, this will be a hierarchical list.

Author(s)

Duncan Temple Lang

See Also

getHTMLExternalFiles

Examples

```
f = system.file("exampleData", "xinclude", "a.xml", package = "XML")

getXIncludes(f, recursive = FALSE)
```

---

getXMLErrors *Get XML/HTML document parse errors*

Description

This function is intended to be a convenience for finding all the errors in an XML or HTML document due to being malformed, i.e. missing quotes on attributes, non-terminated elements/nodes, incorrectly terminated nodes, missing entities, etc. The document is parsed and a list of the errors is returned along with information about the file, line and column number.

Usage

```
getXMLErrors(filename, parse = xmlParse, ...)
```

Arguments

filename — the identifier for the document to be parsed, one of a local file name, a URL or the XML/HTML content itself

parse — the function to use to parse the document, usually either xmlTreeParse or htmlTreeParse.

... — additional arguments passed to the function given by parse

Value

A list of S3-style `XMLError` objects.

Author(s)

Duncan Temple Lang

References

libxml2 (http://xmlsoft.org)

See Also

`error` argument for `xmlTreeParse` and related functions.

Examples

```
   # Get the "errors" in the HTML that was generated from this Rd file
  getXMLErrors(system.file("html", "getXMLErrors.html", package = "XML"))

## Not run:
  getXMLErrors("https://www.omegahat.net/index.html")

## End(Not run)
```

`isXMLString` *Facilities for working with XML strings*

Description

These functions and classes are used to represent and parse a string whose content is known to be XML. `xml` allows us to mark a character vector as containing XML, i.e. of class `XMLString`.

`xmlParseString` is a convenience routine for converting an XML string into an XML node/tree.

`isXMLString` is examines a strings content and heuristically determines whether it is XML.

Usage

```
isXMLString(str)
xmlParseString(content, doc = NULL, namespaces = RXMLNamespaces,
               clean = TRUE, addFinalizer = NA)
xml(x)
```

Arguments

str, x, content	the string containing the XML material.
doc	if specified, an XMLInternalDocument object which is used to "house" the new nodes. Specifically, when the nodes are created, they are made as part of this document. This may not be as relevant now with the garbage collection being done at a node and document level. But it still potentially of some value.
namespaces	a character vector giving the URIs for the XML namespaces which are to be removed if clean is TRUE.
clean	a logical value that controls whether namespaces are removed after the document is parsed..
addFinalizer	a logical value or identifier for a C routine that controls whether we register finalizers on the intenal node.

Value

isXMLString returns a logical value.

xmlParseString returns an object of class XMLInternalElementNode.

xml returns an object of class XMLString identifying the text as XML.

Author(s)

Dncan Temple Lang

See Also

xmlParse xmlTreeParse

Examples

```
isXMLString("a regular string < 20 characters long")
isXMLString("<a><b>c</b></a>")

xmlParseString("<a><b>c</b></a>")

 # We can lie!
isXMLString(xml("foo"))
```

---

length.XMLNode *Determine the number of children in an XMLNode object.*

Description

This function is a simple way to compute the number of sub-nodes (or children) an XMLNode object possesses. It is provided as a convenient form of calling the xmlSize function.

Usage

```
## S3 method for class 'XMLNode'
length(x)
```

Arguments

x	the XMLNode object whose length is to be queried.

Value

An integer giving the number of sub-nodes of this node.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlSize xmlChildren

Examples

```
doc <- xmlTreeParse(system.file("exampleData", "mtcars.xml", package="XML"))
r <- xmlRoot(doc, skip=TRUE)
length(r)
  # get the last entry
r[[length(r)]]
```

---

libxmlVersion *Query the version and available features of the libxml library.*

Description

libxmlVersion retrieves the version of the libxml library used when installing this XML package. libxmlFeatures returns a named logical vector indicating which features are enabled.

Usage

```
libxmlVersion(runTime = FALSE)
libxmlFeatures()
```

Arguments

runTime	a logical value indicating whether to retrieve the version information describing libxml when the R package was compiled or the run-time version. These may be different if a) a new version of libxml2 is installed after the package is installed, b) if the package was installed as a binary package built on a different machine.

Value

libxmlVersion returns a named list with fields

major	the major version number, either 1 or 2 indicating the old or new-style library.
minor	the within version release number.
patch	the within minor release version number

libxmlFeatures returns a logical vector with names given by: [1] "THREAD" "TREE" "OUTPUT" "PUSH" "READER" [6] "PATTERN" "WRITER" "SAX1" "FTP" "HTTP" [11] "VALID" "HTML" "LEGACY" "C14N" "CATALOG" [16] "XPATH" "XPTR" "XINCLUDE" "ICONV" "ISO8859X" [21] "UNICODE" "REGEXP" "AUTOMATA" "EXPR" "SCHEMAS" [26] "SCHEMATRON" "MODULES" "DEBUG" "DEBUG_MEM" "DEBUG_RUN" [31] "ZLIB" Elements are either TRUE or FALSE indicating whether support was activatd for that feature, or NA if that feature is not part of the particular version of libcurl.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.xmlsoft.org

Examples

```
ver <- libxmlVersion()
if(is.null(ver)) {
  cat("Relly old version of libxml\n")
} else {
  if(ver$major > 1) {
    cat("Using libxml2\n")
  }
}
```

---

makeClassTemplate *Create S4 class definition based on XML node(s)*

Description

This function is used to create an S4 class definition by examining an XML node and mapping the sub-elements to S4 classes. This works very simply with child nodes being mapped to other S4 classes that are defined recursively in the same manner. Simple text elements are mapped to a generic character string. Types can be mapped to more specific types (e.g. boolean, Date, integer) by the caller (via the `types`) parameter. The function also generates a coercion method from an `XMLAbstractNode` to an instance of this new class.

This function can either return the code that defines the class or it can define the new class in the R session.

Usage

```
makeClassTemplate(xnode, types = character(), default = "ANY",
                  className = xmlName(xnode), where = globalenv())
```

Arguments

`xnode`	the XML node to analyze
`types`	a character vector mapping XML elements to R classes
`default`	the default class to map an element to
`className`	the name of the new top-level class to be defined. This is the name of the XML node (without the name space)
`where`	typically either an environment or NULL. This is used to control where the class and coercion method are defined or if `NULL` inhibits the code being evaluated. In this case, the code is returned as strings.

Value

A list with 4 elements:

`name`	the name of the new class
`slots`	a character vector giving the slot name and type name pairs
`def`	code for defining the class
`coerce`	code for defining the coercion method from an XMLAbstractNode to an instance of the new class

If `where` is not `NULL`, the class and coercion code is actually evaluated and the class and method will be defined in the R session as a side effect.

Author(s)

Duncan Temple Lang

See Also

`xmlToS4`

Examples

```
txt = paste0("<doc><part><name>ABC</name><type>XYZ</type>",
            "<cost>3.54</cost><status>available</status></part></doc>")
 doc = xmlParse(txt)

 code = makeClassTemplate(xmlRoot(doc)[[1]], types = c(cost = "numeric"))

 as(xmlRoot(doc)[["part"]], "part")
```

`names.XMLNode` *Get the names of an XML nodes children.*

Description

This is a convenient way to obtain the XML tag name of each of the sub-nodes of a given `XMLNode` object.

Usage

```
## S3 method for class 'XMLNode'
names(x)
```

Arguments

`x`	the `XMLNode` whose sub-node tag names are being queried.

Value

A character vector returning the tag names of the sub-nodes of the given `XMLNode` argument.

Note

This overrides the regular names method which would display the names of the internal fields of an `XMLNode` object. Since these are intended to be invisible and queried via the accessor methods (`xmlName`, `xmlAttrs`, etc.), this should not be a problem. If you really need the names of the fields, use `names(unclass(x))`.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

`xmlApply` `xmlSApply`

Examples

```
doc <- xmlTreeParse(system.file("exampleData", "mtcars.xml", package="XML"))
names(xmlRoot(doc))

r <- xmlRoot(doc)
r[names(r) == "variables"]
```

---

newXMLDoc *Create internal XML node or document object*

Description

These are used to create internal 'libxml' nodes and top-level document objects that are used to write XML trees. While the functions are available, their direct use is not encouraged. Instead, use xmlTree as the functions need to be used within a strict regime to avoid corrupting C level structures.

xmlDoc creates a new XMLInternalDocument object by copying the given node and all of its descendants and putting them into a new document. This is useful when we want to work with sub-trees with general tools that work on documents, e.g. XPath queries.

newXMLDoc allows one to create a regular XML node with a name and attributes. One can provide new namespace definitions via namespaceDefinitions. While these might also be given in the attributes in the slightly more verbose form of c('xmlns:prefix' = 'http://...'), the result is that the XML node does not interpret that as a namespace definition but merely an attribute with a name 'xmlns:prefix'. Instead, one should specify the namespace definitions via the namespaceDefinitions parameter.

In addition to namespace definitions, a node name can also have a namespace definition. This can be specified in the name argument as prefix:name and newXMLDoc will do the right thing in separating this into the namespace and regular name. Alternatively, one can specify a namespace separately via the namespace argument. This can be either a simple name or an internal namespace object defined earlier.

How do we define a default namespace?

Usage

```
xmlDoc(node, addFinalizer = TRUE)
newXMLDoc(dtd = "", namespaces=NULL, addFinalizer = TRUE,
           name = character(), node = NULL, isHTML = FALSE)
newHTMLDoc(dtd = "loose", addFinalizer = TRUE, name = character(),
            node = newXMLNode("html",
                              newXMLNode("head", addFinalizer = FALSE),
                              newXMLNode("body", addFinalizer = FALSE),
                             addFinalizer = FALSE))
newXMLNode(name, ..., attrs = NULL, namespace = character(),
            namespaceDefinitions = character(),
             doc = NULL, .children = list(...), parent = NULL,
```

```
      at = NA, cdata = FALSE,
             suppressNamespaceWarning =
                 getOption("suppressXMLNamespaceWarning", FALSE),
             sibling = NULL, addFinalizer = NA,
              noNamespace = length(namespace) == 0 && !missing(namespace),
               fixNamespaces = c(dummy = TRUE, default = TRUE))
newXMLTextNode(text, parent = NULL, doc = NULL, cdata = FALSE,
                escapeEntities = is(text, "AsIs"), addFinalizer = NA)
newXMLCDataNode(text, parent = NULL, doc = NULL, at = NA, sep = "\n",
                   addFinalizer = NA)
newXMLCommentNode(text, parent = NULL, doc = NULL, at = NA, addFinalizer = NA)
newXMLPINode(name, text, parent = NULL, doc = NULL, at = NA, addFinalizer = NA)
newXMLDTDNode(nodeName, externalID = character(),
              systemID = character(), doc = NULL, addFinalizer = NA)
```

Arguments

node	a XMLInternalNode object that will be copied to create a subtree for a new document.
dtd	the name of the DTD to use for the XML document. Currently ignored!
namespaces	a named character vector with each element specifying a name space identifier and the corresponding URI for that namespace that are to be declared and used in the XML document, \e.g. c(shelp = "https://www.omegahat.net/XML/SHelp")
addFinalizer	a logical value indicating whether the default finalizer routine should be registered to free the internal xmlDoc when R no longer has a reference to this external pointer object. This can also be the name of a C routine or a reference to a C routine retrieved using getNativeSymbolInfo.
name	the tag/element name for the XML node and the for a Processing Instruction (PI) node, this is the "target", e.g. the identifier for the system for whose attention this PI node is intended.
...	the children of this node. These can be other nodes created earlier or R strings that are converted to text nodes and added as children to this newly created node.
attrs	a named list of name-value pairs to be used as attributes for the XML node. One should not use this argument to define namespaces, i.e. attributes of the form xmlns:prefix='http:/...'. Instead, such definitions should be specified ideally via the namespaceDefinitions argument, or even the namespace argument. The reason is that namespace definitions are special attributes that are shared across nodes wherease regular attributes are particular to a node. So a namespace needs to be explicitly defined so that the XML representation can recognize it as such.
namespace	a character vector specifying the namespace for this new node. Typically this is used to specify i) the prefix of the namespace to use, or ii) one or more namespace definitions, or iii) a combination of both. If this is a character vector with a) one element and b) with an empty names attribute and c) whose value does not start with http:/ or ftp:/, then it is assumed that the value is a namespace prefix for a namespace defined in an ancestor node. To be able to resolve this prefix to a namespace definition, parent must be specified so that we can traverse the

chain of ancestor nodes. However, if c) does not hold, i.e. the string starts with `http:/` or `ftp:/`, then we take this single element to be a namespace definition and the since it has no name b), this is the definition for the default namespace for this new node, i.e. corresponding to `xmlns='http:/...'`. It is cumbersome to specify `""` as a name for an element in a character vector (as `c('' = 'value')` gives an unnecessary `error!`. Elements with names are expanded to namespace definitions with the name as the prefix and the value as the namespace URI.

`doc` the `XMLInternalDocument` object created with `newXMLDoc` that is used to root the node.

`.children` a list containing XML node elements or content. This is an alternative form of specifying the child nodes than ... which is useful for programmatic interaction when the "sub"-content is already in a list rather than a loose collection of values.

`text` the text content for the new XML node

`nodeName` the name of the node to put in the DOCTYPE element that will appear as the top-most node in the XML document.

`externalID` the PUBLIC identifier for the document type. This is a string of the form `A//B//C//D`. A is either + or -; B identifies the person or insitution that defined the format (i.e. the "creator"); C is the name of the format; and language is an encoding for the language that comes from the ISO 639 document.

`systemID` the SYSTEM identifier for the DTD for the document. This is a URI

`namespaceDefinitions`
a character vector or a list with each element being a string. These give the URIs identifying the namespaces uniquely. The elements should have names which are used as prefixes. A default namespace has "" as the name. This argument can be used to remove any ambiguity that arises when specifying a single string with no names attribute as the value for `namespace`. The values here are used only for defining new namespaces and not for determining the namespace to use for this particular node.

`parent` the node which will act as the parent of this newly created node. This need not be specified and one can add the new node to another node in a separate operation via `addChildren`.

`sibling` if this is specified (rather than `parent`) this should be an `XMLInternalNode` and the new node is added as a sibling of this node, after this node, i.e. to the right. This is just a convenient form of `parent = xmlParent(node)`.

`cdata` a logical value indicating whether to enclose the text within a CDATA node (`TRUE`) or not (`FALSE`). This is a convenience mechanism to avoid having to create the text node and then the CDATA node. If one is not certain what characters are in the text, it is useful to use `TRUE` to ensure that they are "escaped".

It is an argument for `newXMLNode` as the child nodes can be given as simple strings and are converted to text nodes. This `cdata` value is passed to the calls to create these text nodes and so controls whether they are enclosed within CDATA nodes.

`suppressNamespaceWarning`
see `addChildren`

at	this allows one to control the position in the list of children at which the node should be added. The default means at the end and this can be any position from 0 to the current number of children.
sep	when adding text nodes, this is used as an additional separator text to insert between the specified strings.
escapeEntities	a logical value indicating whether to mark the internal text node in such a way that protects characters in its contents from being escaped as entities when being serialized via saveXML
noNamespace	a logical value that allows the caller to specify that the new node has no namespace. This can avoid searching parent and ancestor nodes up the tree for the default namespace.
isHTML	a logical value that indicates whether the XML document being created is HTML or generic XML. This helps to create an object that is identified as an HTML document.
fixNamespaces	a logical vector controlling how namespaces in child nodes are to be processed. The two entries should be named dummy and default. The dummy element controls whether we process child nodes that have a namespace which was not defined when the node was created. These are created as "dummy" namespaces and can be resolved now that the parent node is defined and the name space may be defined. When we know it is not yet defined, but will be defined in an ancestor node, we can turn off this processing with a value of FALSE. The default element controls how we process the child nodes and give them the default name space defined in the parent or ancestor nodes.

Details

These create internal C level objects/structure instances that can be added to a libxml DOM and subsequently inserted into other document objects or "serialized" to textual form.

Value

Each function returns an R object that points to the C-level structure instance. These are of class XMLInternalDocument and XMLInternalNode, respectively

Note

These functions are used to build up an internal XML tree. This can be used in the Sxslt package when creating content in R that is to be dynamically inserted into an XML document.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.xmlsoft.org

See Also

xmlTree saveXML

Examples

```
doc = newXMLDoc()

 # Simple creation of an XML tree using these functions
top = newXMLNode("a")
newXMLNode("b", attrs = c(x = 1, y = 'abc'), parent = top)
newXMLNode("c", "With some text", parent = top)
d = newXMLNode("d", newXMLTextNode("With text as an explicit node"), parent = top)
newXMLCDataNode("x <- 1\n x > 2", parent = d)

newXMLPINode("R", "library(XML)", top)
newXMLCommentNode("This is a comment", parent = top)

o = newXMLNode("ol", parent = top)

kids = lapply(letters[1:3],
               function(x)
                  newXMLNode("li", x))
addChildren(o, kids)

cat(saveXML(top))


x = newXMLNode("block", "xyz", attrs = c(id = "bob"),
                      namespace = "fo",
                      namespaceDefinitions = c("fo" = "http://www.fo.org"))

xmlName(x, TRUE) == "fo"

  # a short cut to define a name space and make it the prefix for the
  # node, thus avoiding repeating the prefix via the namespace argument.
x = newXMLNode("block", "xyz", attrs = c(id = "bob"),
                      namespace = c("fo" = "http://www.fo.org"))


 # name space on the attribute
x = newXMLNode("block", attrs = c("fo:id" = "bob"),
                      namespaceDefinitions = c("fo" = "http://www.fo.org"))




x = summary(rnorm(1000))
d = xmlTree()
d$addNode("table", close = FALSE)

d$addNode("tr", .children = sapply(names(x), function(x) d$addNode("th", x)))
d$addNode("tr", .children = sapply(x, function(x) d$addNode("td", format(x))))
```

```
d$closeNode()


# Just doctype
z = xmlTree("people", dtd = "people")
# no public element
z = xmlTree("people", dtd = c("people", "", "https://www.omegahat.net/XML/types.dtd"))
# public and system
z = xmlTree("people", dtd = c("people", "//a//b//c//d", "https://www.omegahat.net/XML/types.dtd"))

# Using a DTD node directly.
dtd = newXMLDTDNode(c("people", "", "https://www.omegahat.net/XML/types.dtd"))
z = xmlTree("people", dtd = dtd)


x = rnorm(3)
z = xmlTree("r:data", namespaces = c(r = "http://www.r-project.org"))
z$addNode("numeric", attrs = c("r:length" = length(x)), close = FALSE)
lapply(x, function(v) z$addNode("el", x))
z$closeNode()
# should give   <r:data><numeric r:length="3"/></r:data>


# shows namespace prefix on an attribute, and different from the one on the node.
z = xmlTree()
z$addNode("r:data",
         namespace = c(r = "http://www.r-project.org",
                       omg = "https://www.omegahat.net"),
         close = FALSE)
x = rnorm(3)
z$addNode("r:numeric", attrs = c("omg:length" = length(x)))


z = xmlTree("people", namespaces = list(r = "http://www.r-project.org"))
z$setNamespace("r")

z$addNode("person", attrs = c(id = "123"), close = FALSE)
z$addNode("firstname", "Duncan")
z$addNode("surname", "Temple Lang")
z$addNode("title", "Associate Professor")
z$addNode("expertize", close = FALSE)
z$addNode("topic", "Data Technologies")
z$addNode("topic", "Programming Language Design")
z$addNode("topic", "Parallel Computing")
z$addNode("topic", "Data Visualization")
z$closeTag()
z$addNode("address", "4210 Mathematical Sciences Building, UC Davis")



   #
txt = newXMLTextNode("x &lt; 1")
```

```
txt # okay
saveXML(txt) # x &amp;lt; 1

   # By escaping the text, we ensure the entities don't
   # get expanded, i.e. &lt; doesn't become &amp;lt;
txt = newXMLTextNode(I("x &lt; 1"))
txt # okay
saveXML(txt) # x &lt; 1


newXMLNode("r:expr", newXMLTextNode(I("x < 1")),
            namespaceDefinitions = c(r = "http://www.r-project.org"))
```

`newXMLNamespace` *Add a namespace definition to an XML node*

Description

This function, and associated methods, define a name space `prefix` = `URI` combination for the given XML node. It can also optionally make this name space the default namespace for the node.

Usage

```
newXMLNamespace(node, namespace, prefix = names(namespace), set = FALSE)
```

Arguments

`node`	the XML node for which the name space is to be defined.
`namespace`	the namespace(s). This can be a simple character vector giving the URI, a named character vector giving the prefix = URI pairs, with the prefixes being the names of the character vector, or one or more (a list) of `XMLNamespace` objects, e.g. returned from a call to `xmlNamespaceDefinitions`
`prefix`	the prefixes to be associated with the URIs given in `namespace`.
`set`	a logical value indicating whether to set the namespace for this node to this newly created name space definition.

Value

An name space definition object whose class corresponds to the type of XML node given in `node`.

Note

Currently, this only applies to XMLInternalNodes. This will be rectified shortly and apply to RXMLNode and its non-abstract classes.

Author(s)

Duncan Temple Lang

References

~put references to the literature/web site here ~

See Also

Constructors for different XML node types - newXMLNode xmlNode. newXMLNamespace.

Examples

```
foo = newXMLNode("foo")
ns = newXMLNamespace(foo, "http://www.r-project.org", "r")
as(ns, "character")
```

---

parseURI *Parse a URI string into its elements*

Description

This breaks a URI given as a string into its different elements such as protocol/scheme, host, port, file name, query. This information can be used, for example, when constructing URIs relative to a base URI.

The return value is an S3-style object of class URI.

This function uses libxml routines to perform the parsing.

Usage

```
parseURI(uri)
```

Arguments

uri	a single string

Value

A list with 8 elements

scheme	the name of the protocol being used, http, ftp as a string.
authority	a string represeting a rarely used aspect of URIs
server	a string identifying the host, e.g. r-project.org
user	a string giving the name of the user, e.g. in FTP "ftp://duncan@www.omegahat.net", this would yield "duncan"
path	a string identifying the path of the target file
query	the CGI query part of the string, e.g. the bit after '?' of the form name=value&name=value
fragment	a string giving the coo
port	an integer identifying the port number on which the connection is to be made

See Also

getRelativeURL

Examples

```
## Not run:  ## site is flaky
  parseURI("https://www.omegahat.net:8080/RCurl/index.html")
  parseURI("ftp://duncan@www.omegahat.net:8080/RCurl/index.html")

  parseURI("ftp://duncan@www.omegahat.net:8080/RCurl/index.html#my_anchor")

 as(parseURI("http://duncan@www.omegahat.net:8080/RCurl/index.html#my_anchor"), "character")

 as(parseURI("ftp://duncan@www.omegahat.net:8080/RCurl/index.html?foo=1&bar=axd"), "character")

## End(Not run)
```

---

parseXMLAndAdd *Parse XML content and add it to a node*

Description

This function parses the given XML content as a string by putting it inside a top-level node and then returns the document or adds the children to the specified parent. The motivation for this function is when we can use string manipulation to efficiently create the XML content by using vectorized operations in R, but then converting that content into parsed nodes.

Generating XML/HTML content by glueing strings together is a poor approach. It is often convenient, but rarely good general software design. It makes for bad software that is not very extensible and difficult to maintain and enhance. Structure that it is programmatically accessible is much better. The tree approach provides this structure. Using strings is convenient and somewhat appropriate when done atomically for large amounts of highly regular content. But then the results should be converted to the structured tree so that they can be modified and extended. This function facilitates using strings and returning structured content.

Usage

```
parseXMLAndAdd(txt, parent = NULL, top = "tmp", nsDefs = character())
```

Arguments

txt	the XML content to parse
parent	an XMLInternalNode to which the top-level nodes in txt will be added as children
top	the name for the top-level node. If parent is specified, this is used but irrelevant.
nsDefs	a character vector of name = value pairs giving namespace definitions to be added to the top node.

Value

If `parent` is `NULL`, the root node of the parsed document is returned. This will be an element whose name is given by `top` unless the XML content in `txt` is AsIs or `code` is empty.

If `parent` is non-`NULL`, .

Author(s)

Duncan Temple Lang

See Also

`newXMLNode` `xmlParse` `addChildren`

Examples

```
 long = runif(10000, -122, -80)
 lat = runif(10000, 25, 48)

txt = sprintf("<Placemark><Point><coordinates>%.3f,%.3f,0</coordinates></Point></Placemark>",
               long, lat)
 f = newXMLNode("Folder")
 parseXMLAndAdd(txt, f)
 xmlSize(f)


## Not run:
     # this version is much slower as i) we don't vectorize the
     #  creation of the XML nodes, and ii) the parsing of the XML
     # as a string is very fast as it is done in C.
 f = newXMLNode("Folder")
 mapply(function(a, b) {
         newXMLNode("Placemark",
                    newXMLNode("Point",
                               newXMLNode("coordinates",
                                          paste(a, b, "0", collapse = ","))),
       parent = f)
         },
       long, lat)
 xmlSize(f)


 o = c("<x>dog</x>", "<omg:x>cat</omg:x>")
 node = parseXMLAndAdd(o, nsDefs  = c("http://cran.r-project.org",
                                      omg = "https://www.omegahat.net"))
 xmlNamespace(node[[1]])
 xmlNamespace(node[[2]])

 tt = newXMLNode("myTop")
 node = parseXMLAndAdd(o, tt, nsDefs  = c("http://cran.r-project.org",
                                          omg = "https://www.omegahat.net"))
 tt
```

```
## End(Not run)
```

`print.XMLAttributeDef` *Methods for displaying XML objects*

Description

These different methods attempt to provide a convenient way to display R objects representing XML elements when they are printed in the usual manner on the console, files, etc. via the `print` function. Each typically outputs its contents in the way that they would appear in an XML document.

Usage

```
## S3 method for class 'XMLNode'
print(x, ..., indent= "", tagSeparator = "\n")
## S3 method for class 'XMLComment'
print(x, ..., indent = "", tagSeparator = "\n")
## S3 method for class 'XMLTextNode'
print(x, ..., indent = "", tagSeparator = "\n")
## S3 method for class 'XMLCDataNode'
print(x, ..., indent="", tagSeparator = "\n")
## S3 method for class 'XMLProcessingInstruction'
print(x, ..., indent="", tagSeparator = "\n")
## S3 method for class 'XMLAttributeDef'
print(x, ...)
## S3 method for class 'XMLElementContent'
print(x, ...)
## S3 method for class 'XMLElementDef'
print(x, ...)
## S3 method for class 'XMLEntity'
print(x, ...)
## S3 method for class 'XMLEntityRef'
print(x, ..., indent= "", tagSeparator = "\n")
## S3 method for class 'XMLOrContent'
print(x, ...)
## S3 method for class 'XMLSequenceContent'
print(x, ...)
```

Arguments

`x`	the XML object to be displayed
`...`	additional arguments for controlling the output from print. Currently unused.

`indent` a prefix that is emitted before the node to indent it relative to its parent and child nodes. This is appended with a space at each succesive level of the tree. If no indentation is desired (e.g. when `xmlTreeParse` is called with `trim` and `ignoreBlanks` being `FALSE`) and `TRUE` respectively, one can pass the value `FALSE` for this `indent` argument.

`tagSeparator` when printing nodes, successive nodes and children are by default displayed on new lines for easier reading. One can specify a string for this argument to control how the elements are separated in the output. The primary purpose of this argument is to allow no space between the elements, i.e. a value of `""`.

Value

Currently, `NULL`.

Note

We could make the node classes self describing with information about whether `ignoreBlanks` was `TRUE` or `FALSE` and if trim was TRUE or FALSE. This could then be used to determine the appropriate values for `indent` and `tagSeparator`. Adding an S3 class element would allow this to be done without the addition of an excessive number of classes.

Author(s)

Duncan Temple Lang

References

https://www.w3.org

See Also

`xmlTreeParse`

Examples

```
fileName <- system.file("exampleData", "event.xml", package ="XML")

   # Example of how to get faithful copy of the XML.
doc = xmlRoot(xmlTreeParse(fileName, trim = FALSE, ignoreBlanks = FALSE))
print(doc, indent = FALSE, tagSeparator = "")

   # And now the default mechanism
doc = xmlRoot(xmlTreeParse(fileName))
print(doc)
```

`processXInclude` *Perform the XInclude substitutions*

Description

This function and its methods process the XInclude directives within the document of the form `<xi:include href="..." xpointer=".."` and perform the actual substitution.

These are only relevant for "internal nodes" as generated via `xmlInternalTreeParse` and `newXMLNode` and their related functions. When dealing with XML documents via `xmlTreeParse` or `xmlEventParse`, the XInclude nodes are controlled during the parsing.

Usage

```
processXInclude(node, flags = 0L)
```

Arguments

`node`	an XMLInternalDocument object or an XMLInternalElement node or a list of such internal nodes, e.g. returned from `xpathApply`.
`flags`	an integer value that provides information to control how the XInclude substitutions are done, i.e. how they are parsed. This is a bitwise OR'ing of some or all of the xmlParserOption values. This will be turned into an enum in R in the future.

Value

These functions are used for their side-effect to modify the document and its nodes.

Author(s)

Duncan Temple Lang

References

libxml2 http://www.xmlsoft.org XInclude

See Also

`xmlInternalTreeParse` `newXMLNode`

Examples

```
f = system.file("exampleData", "include.xml", package = "XML")
doc = xmlInternalTreeParse(f, xinclude = FALSE)

cat(saveXML(doc))
sects = getNodeSet(doc, "//section")
sapply(sects, function(x) xmlName(x[[2]]))
```

```
processXInclude(doc)

cat(saveXML(doc))

f = system.file("exampleData", "include.xml", package = "XML")
doc = xmlInternalTreeParse(f, xinclude = FALSE)
section1 = getNodeSet(doc, "//section")[[1]]

   # process
processXInclude(section1[[2]])
```

`readHTMLList` *Read data in an HTML list or all lists in a document*

Description

This function and its methods are somewhat similar to `readHTMLTable` but read the contents of lists in an HTML document. We can specify the URL of the document or an already parsed document or an individual node within the document.

Usage

```
readHTMLList(doc, trim = TRUE, elFun = xmlValue, which = integer(), ...)
```

Arguments

`doc`	the URL of the document or the parsed HTML document or an individual node.
`trim`	a logical value indicating whether we should remove leading and trailing white space in each list item when returning it
`elFun`	a function that is used to process each list item node (`li`). This provides an opportunity to customize how each node is processed, for example accessing attributes on the list item or on its contents such as links in the items.
`which`	an index or name which or vector of same which identifies which list nodes to process in the overall document. This is for subsetting particular lists rather than processing them all.
`...`	additional arguments passed to `htmlParse` and for the specific methods.

Value

A list of character vectors or lists, with one element for each list in the document. If only one list is being read (by specifying `which` as a single identifier), that is returned as is.

Author(s)

Duncan Temple Lang

See Also

readHTMLTable

Examples

```
## Not run:
  try(readHTMLList("https://www.omegahat.net"))

## End(Not run)
```

---

readHTMLTable *Read data from one or more HTML tables*

Description

This function and its methods provide somewhat robust methods for extracting data from HTML tables in an HTML document. One can read all the tables in a document given by filename or (http: or ftp:) URL, or having already parsed the document via htmlParse. Alternatively, one can specify an individual <table> node in the document.

The methods attempt to do some heuristic computations to determine the header labels for the columns, the name of the table, etc.

Usage

```
readHTMLTable(doc, header = NA,
              colClasses = NULL, skip.rows = integer(), trim = TRUE,
              elFun = xmlValue, as.data.frame = TRUE, which = integer(),
               ...)
```

Arguments

doc	the HTML document which can be a file name or a URL or an already parsed HTMLInternalDocument, or an HTML node of class XMLInternalElementNode, or a character vector containing the HTML content to parse and process.
header	either a logical value indicating whether the table has column labels, e.g. the first row or a thead, or alternatively a character vector giving the names to use for the resulting columns. This can be a logical vector and the individual values will be used in turn for the different tables. This allows the caller to control whether individual tables are processed as having column names. Alternatively, one can read a specific table via the which parameter and control how that is processed with a single scalar logical.
colClasses	either a list or a vector that gives the names of the data types for the different columns in the table, or alternatively a function used to convert the string values to the appropriate type. A value of NULL means that we should drop that column from the result. Note that currently the conversion occurs before the vectors are converted to a data frame (if as.data.frame is TRUE). As a result, to ensure that

character vectors remain as characters and not factors, use `stringsAsFactors = FALSE`. This typically applies only to an individual table and so for the method applied to a `XMLInternalElementNode` object.

In addition to the usual "integer", "numeric", "logical", "character", etc. names of R data types, one can use "FormattedInteger", "FormattedNumber" and "Percent" to specify that format of the values are numbers possibly with commas (,) separating groups of digits or a number followed by a percent sign (%). This mechanism allows one to introduce new classes and specify these as targets in `colClasses`.

`skip.rows` an integer vector indicating which rows to ignore.

`trim` a logical value indicating whether to remove leading and trailing white space from the content cells.

`elFun` a function which, if specified, is called when converting each cell. Currently, only the node is specified. In the future, we might additionally pass the index of the column so that the function has some context, e.g. whether the value is a row label or a regular value, or if the caller knows the type of columns.

`as.data.frame` a logical value indicating whether to turn the resluting table(s) into data frames or leave them as matrices.

`which` an integer vector identifying which tables to return from within the document. This applies to the method for the document, not individual tables.

`...` currently additional parameters that are passed on to `as.data.frame` if `as.data.frame` is `TRUE`. We may change this to use these as additional arguments for calls to `elFun`.

Value

If the document (either by name or parsed tree) is specified, the return vale is a list of data frames or matrices. If a single HTML node is provided

Author(s)

Duncan Temple Lang

References

HTML4.0 specification

See Also

`htmlParse` `getNodeSet` `xpathSApply`

Examples

```
## Not run:
## This changed to using https: in June 2015, and that is unsupported.
# u = "http://en.wikipedia.org/wiki/World_population"
 u = "https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population"
```

```
tables = readHTMLTable(u)
names(tables)

tables[[2]]
 # Print the table. Note that the values are all characters
 # not numbers. Also the column names have a preceding X since
 # R doesn't allow the variable names to start with digits.
tmp = tables[[2]]


  # Let's just read the second table directly by itself.
doc = htmlParse(u)
tableNodes = getNodeSet(doc, "//table")
tb = readHTMLTable(tableNodes[[2]])

 # Let's try to adapt the values on the fly.
 # We'll create a function that turns a th/td node into a val
tryAsInteger = function(node) {
                 val = xmlValue(node)
                 ans = as.integer(gsub(",", "", val))
                 if(is.na(ans))
                     val
                 else
                     ans
               }

tb = readHTMLTable(tableNodes[[2]], elFun = tryAsInteger)

tb = readHTMLTable(tableNodes[[2]], elFun = tryAsInteger,
                      colClasses = c("character", rep("integer", 9)))

## End(Not run)

zz =
 readHTMLTable("https://www.inflationdata.com/Inflation/Consumer_Price_Index/HistoricalCPI.aspx")
if(any(i <- sapply(zz, function(x) if(is.null(x)) 0 else ncol(x)) == 14)) {
  # guard against the structure of the page changing.
    zz = zz[[which(i)[1]]]  # 4th table
    # convert columns to numeric.  Could use colClasses in the call to readHTMLTable()
    zz[-1] = lapply(zz[-1], function(x) as.numeric(gsub(".* ", "", as.character(x))))
    matplot(1:12, t(zz[-c(1, 14)]), type = "l")
}


# From Marsh Feldman on R-help, possibly
# https://stat.ethz.ch/pipermail/r-help/2010-March/232586.html
# That site was non-responsive in June 2015,
# and this does not do a good job on the current table.

## Not run:
doc <- "http://www.nber.org/cycles/cyclesmain.html"
# The  main table is the second one because it's embedded in the page table.
tables <- getNodeSet(htmlParse(doc), "//table")
```

```
xt <- readHTMLTable(tables[[2]],
                    header = c("peak","trough","contraction",
                               "expansion","trough2trough","peak2peak"),
                    colClasses = c("character","character","character",
                                   "character","character","character"),
                    trim = TRUE, stringsAsFactors = FALSE
                   )

## End(Not run)
if(FALSE) {
 # Here is a totally different way of reading tables from HTML documents.
 # The data are formatted using PRE and so can be read via read.table
 u = "http://tidesonline.nos.noaa.gov/data_read.shtml?station_info=9414290+San+Francisco,+CA"
 h = htmlParse(u)
 p = getNodeSet(h, "//pre")
 con = textConnection(xmlValue(p[[2]]))
 tides = read.table(con)
}

## Not run:
## This is not accessible without authentication ...
u = "https://www.omegahat.net/RCurl/testPassword/table.html"
if(require(RCurl) && url.exists(u)) {
  tt =  getURL(u, userpwd = "bob:duncantl")
  readHTMLTable(tt)
}
## End(Not run)
```

`readKeyValueDB` *Read an XML property-list style document*

Description

This function and its methods reads an XML document that is in the format of name-value or key-value pairs made up of a `plist` and `dict` nodes, each of which is made up `key`, and value node pairs. These used to be used for property lists on OS X and can represetn arbitrary data relatively conveniently.

Usage

```
readKeyValueDB(doc, ...)
```

Arguments

`doc`	the object containing the data. This can be the name of a file, a parsed XML document or an XML node.
`...`	additional parameters for the methods. One can pass `dropComments` as a logical value to control whether comment nodes are processed or ignored (`TRUE`).

Value

An R object representing the data read from the XML content. This is typically a named list or vector where the names are the keys and the values are collected into an R "container".

Author(s)

Duncan Temple Lang

References

Property lists.

See Also

readSolrDoc, xmlToList, xmlToDataFrame, xmlParse

Examples

```
if(file.exists("/usr/share/hiutil/Stopwords.plist")) {
 o = readKeyValueDB("/usr/share/hiutil/Stopwords.plist")
}

if(file.exists("/usr/share/java/Tools/Applet Launcher.app/Contents/Info.plist"))
  javaInfo = readKeyValueDB('/usr/share/java/Tools/Applet Launcher.app/Contents/Info.plist')
```

---

readSolrDoc *Read the data from a Solr document*

Description

Solr documents are used to represent general data in a reasonably simple format made up of lists, integers, logicals, longs, doubles, dates, etc. each with an optional name. These correspond very naturally to R objects.

Usage

```
readSolrDoc(doc, ...)
```

Arguments

doc	the object containing the data. This can be the name of a file, a parsed XML document or an XML node.
...	additional parameters for the methods.

Value

An R object representing the data in the Solr document, typically a named vector or named list.

Author(s)

Duncan Temple Lang

References

Lucene text search system.

See Also

readKeyValueDB, xmlToList, xmlToDataFrame, xmlParse

Examples

```
f = system.file("exampleData", "solr.xml", package = "XML")
readSolrDoc(f)
```

---

removeXMLNamespaces *Remove namespace definitions from a XML node or document*

Description

This function and its methods allow one to remove one or more XML namespace definitions on XML nodes within a document.

Usage

```
removeXMLNamespaces(node, ..., all = FALSE, .els = unlist(list(...)))
```

Arguments

node	an XMLInternalNode or XMLInternalDocument object
...	the names of the namespaces to remove or an XMLNamespaceRef object returned via getNodeSet or xpathApply.
all	a logical value indicating whether to remove all the namespace definitions on a node.
.els	a list which is sometimes a convenient way to specify the namespaces to remove.

Value

This function is used for its side-effects and changing the internal node.

Author(s)

Duncan Temple Lang

See Also

newXMLNamespace

replaceNodeWithChildren

Replace an XML node with it child nodes

Description

This function can be used to flatten parts of an XML tree. This takes a node and removes itself from the tree, but places its kids in it place.

Usage

```
replaceNodeWithChildren(node)
```

Arguments

node	an XMLInternalNode object

Value

NULL. The purpose of this function is to modify the internal document.

Author(s)

Duncan Temple Lang

References

libxml2 documentation.

Examples

```
doc = xmlParse('<doc>
                 <page>
                  <p>A</p>
                  <p>B</p>
                  <p>C</p>
                 </page>
                 <page>
                  <p>D</p>
                  <p>E</p>
                  <p>F</p>
                 </page>
                </doc>')

pages = getNodeSet(doc, "//page")
invisible(lapply(pages, replaceNodeWithChildren))
doc
```

saveXML *Output internal XML Tree*

Description

Methods for writing the representation of an XML tree to a string or file. Originally this was intended to be used only for DOMs (Document Object Models) stored in internal memory created via xmlTree, but methods for XMLNode, XMLInternalNode and XMLOutputStream objects (and others) allow it to be generic for different representations of the XML tree.

Note that the indentation when writing an internal C-based node (XMLInternalNode) may not be as expected if there are text nodes within the node.

Also, not all the parameters are meaningful for all methods. For example, compressing when writing to a string is not supported.

Usage

```
saveXML(doc, file=NULL, compression=0, indent=TRUE, prefix = '<?xml version="1.0"?>\n',
        doctype = NULL, encoding = getEncoding(doc), ...)
## S3 method for class 'XMLInternalDocument'
saveXML(doc, file=NULL, compression=0, indent=TRUE, prefix = '<?xml version="1.0"?>\n',
                            doctype = NULL, encoding =  getEncoding(doc), ...)
## S3 method for class 'XMLInternalDOM'
saveXML(doc, file=NULL, compression=0, indent=TRUE, prefix = '<?xml version="1.0"?>\n',
                        doctype = NULL, encoding =  getEncoding(doc), ...)
## S3 method for class 'XMLNode'
saveXML(doc, file=NULL, compression=0, indent=TRUE, prefix = '<?xml version="1.0"?>\n',
                  doctype = NULL, encoding = getEncoding(doc), ...)
## S3 method for class 'XMLOutputStream'
saveXML(doc, file=NULL, compression=0, indent=TRUE, prefix = '<?xml version="1.0"?>\n',
                          doctype = NULL, encoding = getEncoding(doc), ...)
```

Arguments

doc	the document object representing the XML document.
file	the name of the file to which the contents of the XML nodes will be serialized.
compression	an integer value between 0 and 9 indicating the level of compression to use when saving the file. Higher values indicate increased compression and hence smaller files at the expense of computational time to do the compression and decompression.
indent	a logical value indicating whether to indent the nested nodes when serializing to the stream.
prefix	a string that is written to the stream/connection before the XML is output. If this is NULL, it is ignored. This allows us to put the XML introduction/preamble at the beginning of the document while allowing it to be omitted when we are outputting multiple "documents" within a single stream.

doctype	an object identifying the elements for the DOCTYPE in the output. This can be a string or an object of class Doctype.
encoding	a string indicating which encoding style to use. This is currently ignored except in the method in Sxslt for saving a document generated by applying an XSL style sheet to an XML document.
...	extra parameters for specific methods

Details

One can create an internal XML tree (or DOM) using newXMLDoc and newXMLNode. saveXML allows one to generate a textual representation of that DOM in human-readable and reusable XML format. saveXML is a generic function that allows one to call the rendering operation with either the top-level node of the DOM or of the document object (of class XMLInternalDocument that is used to accumulate the nodes and with which the developer adds nodes.

Value

If file is not specified, the result is a character string containing the resulting XML content. If file is passed in the call,

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

newXMLDoc newXMLNode xmlOutputBuffer xmlOutputDOM

Examples

```
b = newXMLNode("bob")
saveXML(b)

f = tempfile()
saveXML(b, f)
doc = xmlInternalTreeParse(f)
saveXML(doc)


con <- xmlOutputDOM()
con$addTag("author", "Duncan Temple Lang")
con$addTag("address",  close=FALSE)
con$addTag("office", "2C-259")
con$addTag("street", "Mountain Avenue.")
con$addTag("phone", close=FALSE)
con$addTag("area", "908", attrs=c(state="NJ"))
con$addTag("number", "582-3217")
```

```
con$closeTag() # phone
con$closeTag() # address

saveXML(con$value(), file=file.path(tempdir(), "out.xml"))


# Work with entities

 f = system.file("exampleData", "test1.xml", package = "XML")
 doc = xmlRoot(xmlTreeParse(f))
 outFile = tempfile()
 saveXML(doc, outFile)
 alt = xmlRoot(xmlTreeParse(outFile))
 if(! identical(doc, alt) )
  stop("Problems handling entities!")

 con = textConnection("test1.xml", "w")
 saveXML(doc, con)
 close(con)
 alt = get("test1.xml")
 identical(doc, alt)



 x = newXMLNode("a", "some text", newXMLNode("c", "sub text"), "more text")

 cat(saveXML(x), "\n")

 cat(as(x, "character"), "\n")


    # Showing the prefix parameter
  doc = newXMLDoc()
  n = newXMLNode("top", doc = doc)
  b = newXMLNode("bar", parent = n)

    # suppress the <?xml ...?>
  saveXML(doc, prefix = character())

    # put our own comment in
  saveXML(doc, prefix = "<!-- This is an alternative prefix -->")
    # or use a comment node.
  saveXML(doc, prefix = newXMLCommentNode("This is an alternative prefix"))
```

SAXState-class *A virtual base class defining methods for SAX parsing*

Description

This is a degenerate virtual class which others are expected to sub-class when they want to use S4 methods as handler functions for SAX-based XML parsing. The idea is that one can pass both i)

a collection of handlers to xmlEventParse which are simply the generic functions for the different SAX actions, and ii) a suitable object to maintain state across the different SAX calls. This is used to perform the method dispatching to get the appropriate behavior for the action. Each of these methods is expected to return the updated state object and the SAX parser will pass this in the next callback.

We define this class here so that we can provide default methods for each of the different handler actions. This allows other programmers to define new classes to maintain state that are sub-class of SAXState and then they do not have to implement methods for each of the different handlers.

Objects from the Class

A virtual Class: No objects may be created from it.

Methods

comment.SAX signature(content = "ANY", .state = "SAXState"): ...

endElement.SAX signature(name = "ANY", .state = "SAXState"): ...

entityDeclaration.SAX signature(name = "ANY", base = "ANY", sysId = "ANY", publicId = "ANY", notationName = "ANY", .state = "SAXState"): ...

processingInstruction.SAX signature(target = "ANY", content = "ANY", .state = "SAXState"): ...

startElement.SAX signature(name = "ANY", atts = "ANY", .state = "SAXState"): ...

text.SAX signature(content = "ANY", .state = "SAXState"): ...

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.xmlsoft.org

See Also

xmlEventParse

Examples

```
# For each element in the document, grab the node name
# and increment the count in an vector for this name.

# We define an S4 class named ElementNameCounter which
# holds the vector of frequency counts for the node names.

 setClass("ElementNameCounter",
             representation(elements = "integer"), contains = "SAXState")

# Define a method for handling the opening/start of any XML node
# in the SAX streams.
```

```
setMethod("startElement.SAX",  c(.state = "ElementNameCounter"),
          function(name, atts, .state = NULL) {

            if(name %in% names(.state@elements))
                .state@elements[name] = as.integer(.state@elements[name] + 1)
            else
                .state@elements[name] = as.integer(1)
            .state
          })

filename = system.file("exampleData", "eurofxref-hist.xml.gz", package = "XML")

# Parse the file, arranging to have our startElement.SAX method invoked.
z = xmlEventParse(filename, genericSAXHandlers(),
                  state = new("ElementNameCounter"), addContext = FALSE)

z@elements

 # Get the contents of all the comments in a character vector.

setClass("MySAXState",
            representation(comments = "character"), contains = "SAXState")

setMethod("comment.SAX",  c(.state = "MySAXState"),
          function(content, .state = NULL) {
            cat("comment.SAX called for MySAXState\n")
            .state@comments <- c(.state@comments, content)
            .state
          })

filename = system.file("exampleData", "charts.svg", package = "XML")
st = new("MySAXState")
z = xmlEventParse(filename, genericSAXHandlers(useDotNames = TRUE), state = st)
z@comments
```

---

schema-class *Classes for working with XML Schema*

Description

These are classes used when working with XML schema and using them to validate a document or querying the schema for its elements. The basic representation is an external/native object stored in the `ref` slot.

See Also

xmlSchemaValidate

setXMLNamespace *Set the name space on a node*

Description

This function sets the name space for an XML node, typically an internal node. We can use it to either define a new namespace and use that, or refer to a name space definition in an ancestor of the current node.

Usage

```
setXMLNamespace(node, namespace, append = FALSE)
```

Arguments

node	the node on which the name space is to be set
namespace	the name space to use for the node. This can be a name space prefix (string) defined in an ancestor node, or a named character vector of the form `c(prefix = URI)` that defines a new namespace on this node, or we can use a name space object created with newXMLNamespace.
append	currently ignored.

Value

An object of class `XMLNamespaceRef` which is a reference to the native/internal/C-level name space object.

Author(s)

Duncan Temple Lang

See Also

newXMLNamespace

removeXMLNamespaces

Examples

```
 # define a new namespace
e = newXMLNode("foo")
setXMLNamespace(e, c("r" = "http://www.r-project.org"))

 # use an existing namespace on an ancestor node
e = newXMLNode("top", namespaceDefinitions = c("r" = "http://www.r-project.org"))
setXMLNamespace(e, "r")
e
```

`startElement.SAX` *Generic Methods for SAX callbacks*

Description

This is a collection of generic functions for which one can write methods so that they are called in repsonse to different SAX events. The idea is that one defines methods for different classes of the `.state` argument and dispatch to different methods based on that argument. The functions represent the different SAX events.

Usage

```
startElement.SAX(name, atts, .state = NULL)
endElement.SAX(name, .state = NULL)
comment.SAX(content, .state = NULL)
processingInstruction.SAX(target, content, .state = NULL)
text.SAX(content, .state = NULL)
entityDeclaration.SAX(name, base, sysId, publicId, notationName,  .state = NULL)
.InitSAXMethods(where = "package:XML")
```

Arguments

`name`	the name of the XML element or entity being declared
`atts`	named character vector of XML attributes
`content`	the value/string in the processing instruction or comment
`target`	the target of the processing instruction, e.g. the R in `<?R....>`
`base`	x
`sysId`	the system identifier for this entity
`publicId`	the public identifier for the entity
`notationName`	name of the notation specification
`.state`	the state object on which the user-defined methods should dispatch.
`where`	the package in which the class and method definitions should be defined. This is almost always unspecified.

Value

Each method should return the (potentially modified) state value.

Note

This no longer requires the Expat XML parser to be installed. Instead, we use libxml's SAX parser.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.xmlsoft.org

See Also

xmlEventParse

---

supportsExpat *Determines which native XML parsers are being used.*

Description

Use of the Gnome libxml and Expat parsers is supported in this R/S XML package, but both need not be used when compiling the package. These functions determine whether each is available in the underlying native code.

Usage

```
supportsExpat()
supportsLibxml()
```

Details

One might to use different parsers to test validity of a document in different ways and to get different error messages. Additionally, one parser may be more efficient than the other. These methods allow one to write code in such a way that one parser is preferred and is used if it is available, but the other is used if the first is not available.

Value

Returns TRUE if the corresponding library has been linked into the package.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlEventParse

Examples

```
   # use Expat if possible, otherwise libxml
fileName <- system.file("exampleData", "mtcars.xml", package="XML")
xmlEventParse(fileName, useExpat = supportsExpat())
```

toHTML *Create an HTML representation of the given R object, using internal C-level nodes*

Description

This generic function and the associated methods are intended to create an HTML tree that represents the R object in some intelligent manner. For example, we represent a vector as a table and we represent a matrix also as a table.

Usage

```
toHTML(x, context = NULL)
```

Arguments

x	the R object which is to be represented via an HTML tree
context	an object which provides context in which the node will be used. This is currently arbitrary. It may be used, for example, when creating HTML for R documentation and providing information about variabes and functions that are available on that page and so have internal links.

Details

It would be nicer if we could pass additional arguments to control whether the outer/parent layer is created, e.g. when reusing code for a vector for a row of a matrix.

Value

an object of class `XMLInternalNode`

Author(s)

Duncan Temple Lang

See Also

The `R2HTML` package.

Examples

```
cat(as(toHTML(rnorm(10)), "character"))
```

`toString.XMLNode` *Creates string representation of XML node*

Description

This creates a string from a hierarchical XML node and its children just as it prints on the console or one might see it in a document.

Usage

```
## S3 method for class 'XMLNode'
toString(x, ...)
```

Arguments

`x`	an object of class `XMLNode`.
`...`	currently ignored

Details

This uses a textConnection object using the name .tempXMLOutput. Since this is global, it will overwrite any existing object of that name! As a result, this function cannot be used recursively in its present form.

Value

A character vector with one element, that being the string corresponding to the XML node's contents.

Note

This requires the Expat XML parser to be installed.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

`xmlNode` `xmlTreeParse`

Examples

```
x <- xmlRoot(xmlTreeParse(system.file("exampleData", "gnumeric.xml", package = "XML")))

toString(x)
```

`xmlApply` *Applies a function to each of the children of an XMLNode*

Description

These methods are simple wrappers for the `lapply` and `sapply` functions. They operate on the sub-nodes of the XML node, and not on the fields of the node object itself.

Usage

```
xmlApply(X, FUN, ...)
## S3 method for class 'XMLNode'
xmlApply(X, FUN, ...)
## S3 method for class 'XMLDocument'
xmlApply(X, FUN, ...)
## S3 method for class 'XMLDocumentContent'
xmlApply(X, FUN, ...)
xmlSApply(X, FUN, ...)
## S3 method for class 'XMLNode'
xmlSApply(X, FUN, ...)
## S3 method for class 'XMLDocument'
xmlSApply(X, FUN, ...)
```

Arguments

`X`	the `XMLNode` on whose children the regular `apply` or `sapply` is to be performed
`FUN`	the function to apply to each child node. This is passed directly to the relevant `apply` function.
`...`	additional arguments to be given to each invocation of `FUN`. This is passed directly to the relevant `apply` function.

Value

The result is that obtained from calling the `apply` or `sapply` on `xmlChildren(x)`.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlChildren xmlRoot [.XMLNode sapply lapply

Examples

```
doc <- xmlTreeParse(system.file("exampleData", "mtcars.xml", package="XML"))
r <- xmlRoot(doc)
xmlSApply(r[[2]], xmlName)

xmlApply(r[[2]], xmlAttrs)

xmlSApply(r[[2]], xmlSize)
```

XMLAttributes-class *Class* "XMLAttributes"

Description

A simple class to represent a named character vector of XML attributes some of which may have a namespace. This maintains the name space

Objects from the Class

Objects can be created by calls of the form `new("XMLAttributes", ...)`. These are typically generated via a call to xmlAttrs.

Slots

.Data: Object of class "character"

Extends

Class "character", from data part. Class "vector", by class "character", distance 2. Class "data.frameRowLabels", by class "character", distance 2. Class "SuperClassMethod", by class "character", distance 2.

Methods

[signature(x = "XMLAttributes"): ...

show signature(object = "XMLAttributes"): ...

Author(s)

Duncan Temple Lang

See Also

xmlAttrs newXMLNode xmlParse

Examples

```
nn = newXMLNode("foo", attrs = c(a = "123", 'r:show' = "true"),
                namespaceDefinitions = c(r = "http://www.r-project.org"))
a = xmlAttrs(nn)
a["show"]
```

`xmlAttributeType` *The type of an XML attribute for element from the DTD*

Description

This examines the definition of the attribute, usually returned by parsing the DTD with `parseDTD` and determines its type from the possible values: Fixed, string data, implied, required, an identifier, an identifier reference, a list of identifier references, an entity, a list of entities, a name, a list of names, an element of enumerated set, a notation entity.

Usage

```
xmlAttributeType(def, defaultType=FALSE)
```

Arguments

`def`	the attribute definition object, usually retrieved from the DTD via `parseDTD`.
`defaultType`	whether to return the default value if this attribute is defined as being a value from an enumerated set.

Value

A string identifying the type for the sspecified attributed.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

`parseDTD`

xmlAttrs *Get the list of attributes of an XML node.*

Description

This returns a named character vector giving the name-value pairs of attributes of an XMLNode object which is part of an XML document.

Usage

```
xmlAttrs(node, ...)
'xmlAttrs<-'(node, append = TRUE, suppressNamespaceWarning =
             getOption("suppressXMLNamespaceWarning", FALSE), value)
```

Arguments

node	The XMLNode object whose attributes are to be extracted.
append	a logical value indicating whether to add the attributes in value to the existing attributes within the XML node, or to replace the set of any existing attributes with this new set, i.e. remove the existing ones and then set the attributes with the contents of value.
...	additional arguments for the specific methods. For XML internal nodes, these are addNamespacePrefix and addNamespaceURLs. These are both logical values and indicate whether to prepend the name of the attribute with the namespace prefix and also whether to return the namespace prefix and URL as a vector in the namespaces attribute.
value	a named character vector giving the new attributes to be added to the node.
suppressNamespaceWarning	see addChildren

Value

A named character vector, where the names are the attribute names and the elements are the corresponding values. This corresponds to the (attr<i>, "value<i>") pairs in the XML tag <tag attr1="value1" attr2="value2"

Author(s)

Duncan Temple Lang

References

https://www.w3.org

See Also

xmlChildren, xmlSize, xmlName

Examples

```
fileName <- system.file("exampleData", "mtcars.xml", package="XML")
doc <- xmlTreeParse(fileName)

xmlAttrs(xmlRoot(doc))

xmlAttrs(xmlRoot(doc)[["variables"]])


doc <- xmlParse(fileName)
d = xmlRoot(doc)

xmlAttrs(d)
xmlAttrs(d) <- c(name = "Motor Trend fuel consumption data",
                 author = "Motor Trends")
xmlAttrs(d)

  # clear all the attributes and then set new ones.
removeAttributes(d)
xmlAttrs(d) <- c(name = "Motor Trend fuel consumption data",
                 author = "Motor Trends")


   # Show how to get the attributes with and without the prefix and
   # with and without the URLs for the namespaces.
 doc = xmlParse('<doc xmlns:r="http://www.r-project.org">
                  <el r:width="10" width="72"/>
                  <el width="46"/>
                  </doc>')

 xmlAttrs(xmlRoot(doc)[[1]], TRUE, TRUE)
 xmlAttrs(xmlRoot(doc)[[1]], FALSE, TRUE)
 xmlAttrs(xmlRoot(doc)[[1]], TRUE, FALSE)
 xmlAttrs(xmlRoot(doc)[[1]], FALSE, FALSE)
```

`xmlChildren` *Gets the sub-nodes within an XMLNode object.*

Description

These functions provide access to the children of the given XML node. The simple accessor returns a list of child XMLNode objects within an XMLNode object.

The assignment operator (`xmlChildren<-`) sets the children of the node to the given value and returns the updated/modified node. No checking is currently done on the type and values of the right hand side. This allows the children of the node to be arbitrary R objects. This can be useful but means that one cannot rely on any structure in a node being present..

Usage

```
xmlChildren(x, addNames= TRUE, ...)
```

Arguments

x	an object of class XMLNode.
addNames	a logical value indicating whether to add the XML names of the nodes as names of the R list. This is only relevant for XMLInternalNode objects as XMLNode objects in R already have R-level names.
...	additional arguments for the particular methods, e.g. omitTypes for an XMLInternalNode.

Value

A list whose elements are sub-nodes of the user-specified XMLNode. These are also of class XMLNode.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

xmlChildren,xmlSize, xmlTreeParse

Examples

```
fileName <- system.file("exampleData", "mtcars.xml", package="XML")
doc <- xmlTreeParse(fileName)
names(xmlChildren(doc$doc$children[["dataset"]]))
```

---

xmlCleanNamespaces *Remove redundant namespaces on an XML document*

Description

This is a convenience function that removes redundant repeated namespace definitions in an XML node. It removes namespace definitions in nodes where an ancestor node also has that definition. It does not remove unused namespace definitions.

This uses the NSCLEAN option for xmlParse

Usage

```
xmlCleanNamespaces(doc, options = integer(), out = docName(doc), ...)
```

Arguments

doc	either the name of an XML documentor the XML content itself, or an already parsed document
options	options for the XML parser. NSCLEAN is added to this.
...	additional arguments passed to xmlParse
out	the name of a file to which to write the resulting XML document, or an empty character vector or logical value FALSE to avoid writing the new document.

Value

If the new document is written to a file, the name of the file is returned. Otherwise, the new parsed XML document is returned.

Author(s)

Duncan Temple Lang

References

libxml2 documentation http://xmlsoft.org/html/libxml-parser.html

See Also

xmlParse

Examples

```
f = system.file("exampleData", "redundantNS.xml", package = "XML")
doc = xmlParse(f)
print(doc)
newDoc = xmlCleanNamespaces(f, out = FALSE)
```

---

xmlClone *Create a copy of an internal XML document or node*

Description

These methods allow the caller to create a copy of an XML internal node. This is useful, for example, if we want to use the node or document in an additional context, e.g. put the node into another document while leaving it in the existing document. Similarly, if we want to remove nodes to simplify processing, we probably want to copy it so that the changes are not reflected in the original document.

At present, the newly created object is not garbage collected.

Usage

```
xmlClone(node, recursive = TRUE, addFinalizer = FALSE, ...)
```

Arguments

node	the object to be cloned
recursive	a logical value indicating whether the entire object and all its descendants should be duplicated/cloned (TRUE) or just the top-level object (FALSE)
addFinalizer	typically a logical value indicating whether to bring this new object under R's regular garbage collection. This can also be a reference to a C routine which is to be used as the finalizer. See getNativeSymbolInfo.
...	additional parameters for methods

Value

A new R object representing the object.

Author(s)

Duncan Temple Lang

References

libxml2

See Also

xmlParse newXMLNode newXMLDoc

Examples

```
doc =
xmlParse(paste0('<doc><author id="dtl"><firstname>Duncan</firstname>',
               '<surname>Temple Lang</surname></author></doc>'))

  au = xmlRoot(doc)[[1]]
     # make a copy
  other = xmlClone(au)
     # change it slightly
  xmlAttrs(other) = c(id = "dtl2")
     # add it to the children
  addChildren(xmlRoot(doc), other)
```

XMLCodeFile-class *Simple classes for identifying an XML document containing R code*

Description

These two classes allow the user to identify an XML document or file as containing R code (amongst other content). Objects of either of these classes can then be passed to `source` to read the code into R and also used in `link{xmlSource}` to read just parts of it. `XMLCodeFile` represents the file by its name; `XMLCodeDoc` parses the contents of the file when the R object is created. Therefore, an `XMLCodeDoc` is a snapshot of the contents at a moment in time while an `XMLCodeFile` object re-reads the file each time and so reflects any "asynchronous" changes.

Usage

```
xmlCodeFile(f, parse = FALSE)
```

Arguments

`f`	Path to the XML file.
`parse`	Whether to construct an `XMLCodeDoc` object containing the contents of `f` instead of an `XMLCodeFile` containing its name.

Objects from the Class

One can create these objects using coercion methods, e.g `as("file/name", "XMLCodeFile")` or `as("file/name", "XMLCodeDoc")`. One can also use `xmlCodeFile`.

Slots

`.Data`**:** Object of class `"character"`

Extends

Class `"character"`, from data part. Class `"vector"`, by class "character", distance 2.

Methods

[[`signature(x = "XMLCodeFile", i = "ANY", j = "ANY")`: this method allows one to retrieve/access an individual R code element in the XML document. This is typically done by specifying the value of the XML element's "id" attribute.

coerce `signature(from = "XMLCodeFile", to = "XMLCodeDoc")`: parse the XML document from the "file" and treat the result as a `XMLCodeDoc` object.

source `signature(file = "XMLCodeFile")`: read and evaluate all the R code in the XML document. For more control, use `xmlSource`.

Author(s)

Duncan Temple Lang

See Also

`xmlSource`

Examples

```
src =  system.file("exampleData", "Rsource.xml", package = "XML")
  # mark the string as an XML file containing R code
k = xmlCodeFile(src)

  # read and parse the code, but don't evaluate it.
code = xmlSource(k, eval = FALSE)

  # read and evaluate the code in a special environment.
e = new.env()
ans = xmlSource(k, envir = e)
ls(e)
```

`xmlDOMApply` *Apply function to nodes in an XML tree/DOM.*

Description

This recursively applies the specified function to each node in an XML tree, creating a new tree, parallel to the original input tree. Each element in the new tree is the return value obtained from invoking the specified function on the corresponding element of the original tree. The order in which the function is recursively applied is "bottom-up". In other words, function is first applied to each of the children nodes first and then to the parent node containing the newly computed results for the children.

Usage

```
xmlDOMApply(dom, func)
```

Arguments

`dom`	a node in the XML tree or DOM on which to recursively apply the given function. This should not be the `XMLDocument` itself returned from `xmlTreeParse` but an object of class `XMLNode`. This is typically obtained by calling `xmlRoot` on the return value from `xmlTreeParse`.
`func`	the function to be applied to each node in the XML tree. This is passed the node object for the and the return value is inserted into the new tree that is to be returned in the corresponding position as the node being processed. If the return value is `NULL`, this node is dropped from the tree.

Details

This is a native (C code) implementation that understands the structure of an XML DOM returned from `xmlTreeParse` and iterates over the nodes in that tree.

Value

A tree that parallels the structure in the `dom` object passed to it.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlTreeParse

Examples

```
dom <- xmlTreeParse(system.file("exampleData","mtcars.xml", package="XML"))
tagNames <- function() {
   tags <- character(0)
   add <- function(x) {
     if(inherits(x, "XMLNode")) {
       if(is.na(match(xmlName(x), tags)))
          tags <<- c(tags, xmlName(x))
     }

     NULL
   }

   return(list(add=add, tagNames = function() {return(tags)}))
}

h <- tagNames()
xmlDOMApply(xmlRoot(dom), h$add)
h$tagNames()
```

---

xmlElementsByTagName *Retrieve the children of an XML node with a specific tag name*

Description

This returns a list of the children or sub-elements of an XML node whose tag name matches the one specified by the user.

Usage

```
xmlElementsByTagName(el, name, recursive = FALSE)
```

Arguments

el	the node whose matching children are to be retrieved.
name	a string giving the name of the tag to match in each of el's children.
recursive	a logical value. If this is FALSE, the default, only the direct child nodes are searched. Alternatively, if this is TRUE, all sub-nodes at all levels are searched. In other words, we find all descendants of the node el and return a list with the nodes having the given name. The relationship between the nodes in the resulting list cannot be determined. This is a set of nodes. See the note.

Details

This does a simple matching of names and subsets the XML node's children list. If recursive is TRUE, then the function is applied recursively to the children of the given node and so on.

Value

A list containing those child nodes of el whose tag name matches that specified by the user.

Note

The addition of the recursive argument makes this function behave like the getElementsByTagName in other language APIs such as Java, C#. However, one should be careful to understand that in those languages, one would get back a set of node objects. These nodes have references to their parents and children. Therefore one can navigate the tree from each node, find its relations, etc. In the current version of this package (and for the forseeable future), the node set is a "copy" of the nodes in the original tree. And these have no facilities for finding their siblings or parent. Additionally, one can consume a large amount of memory by taking a copy of numerous large nodes using this facility. If one does not modify the nodes, the extra memory may be small. But modifying them means that the contents will be copied.

Alternative implementations of the tree, e.g. using unique identifiers for nodes or via internal data structures from libxml can allow us to implement this function with different semantics, more similar to the other APIs.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

xmlChildren xmlTreeParse

Examples

```
## Not run:
 doc <- xmlTreeParse("https://www.omegahat.net/Scripts/Data/mtcars.xml")
 xmlElementsByTagName(doc$children[[1]], "variable")

## End(Not run)

 doc <- xmlTreeParse(system.file("exampleData", "mtcars.xml", package="XML"))
 xmlElementsByTagName(xmlRoot(doc)[[1]], "variable")
```

---

xmlElementSummary *Frequency table of names of elements and attributes in XML content*

Description

This function is used to get an understanding of the use of element and attribute names in an XML document. It uses a collection of handler functions to gather the information via a SAX-style parser. The distribution of attribute names is done within each "type" of element (i.e. element name)

Usage

```
xmlElementSummary(url, handlers = xmlElementSummaryHandlers(url))
```

Arguments

`url`	the source of the XML content, e.g. a file, a URL, a compressed file, or a character string
`handlers`	the list of handler functions used to collect the information. These are passed to the function `xmlEventParse` as the value for the `handlers` parameter.

Value

A list with two elements

`nodeCounts`	a named vector of counts where the names are the (XML namespace qualified) element names in the XML content
`attributes`	a list with as many elements as there are elements in the `nodeCounts` element of the result. Each element of this sub-list gives the frequency counts for the different attributes seen within the XML elements with that name.

Author(s)

Duncan Temple Lang

See Also

`xmlEventParse`

Examples

```
xmlElementSummary(system.file("exampleData", "eurofxref-hist.xml.gz", package = "XML"))
```

`xmlEventHandler` *Default handlers for the SAX-style event XML parser*

Description

This is a function that returns a closure instance containing the default handlers for use with `xmlEventParse` for parsing XML documents via the SAX-style parsing.

Usage

```
xmlEventHandler()
```

Details

These handlers simply build up the DOM tree and thus perform the same job as `xmlTreeParse`. It is here more as an example, reference and a base that users can extend.

Value

The return value is a list of functions which are used as callbacks by the internal XML parser when it encounters certain XML elements/structures. These include items such as the start of an element, end of an element, processing instruction, text node, comment, entity references and definitions, etc.

`startElement`

`endElement`
`processingInstruction`

`text`

`comment`

`externalEntity`
`entityDeclaration`

`cdata`

`dom`

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlEventParse xmlTreeParse

Examples

```
xmlEventParse(system.file("exampleData", "mtcars.xml", package="XML"),
              handlers=xmlEventHandler())
```

---

xmlEventParse *XML Event/Callback element-wise Parser*

Description

This is the event-driven or SAX (Simple API for XML) style parser which process XML without building the tree but rather identifies tokens in the stream of characters and passes them to handlers which can make sense of them in context. This reads and processes the contents of an XML file or string by invoking user-level functions associated with different components of the XML tree. These components include the beginning and end of XML elements, e.g <myTag x="1"> and </myTag> respectively, comments, CDATA (escaped character data), entities, processing instructions, etc. This allows the caller to create the appropriate data structure from the XML document contents rather than the default tree (see xmlTreeParse) and so avoids having the entire document in memory. This is important for large documents and where we would end up with essentially 2 copies of the data in memory at once, i.e the tree and the R data structure containing the information taken from the tree. When dealing with classes of XML documents whose instances could be large, this approach is desirable but a little more cumbersome to program than the standard DOM (Document Object Model) approach provided by XMLTreeParse.

Note that xmlTreeParse does allow a hybrid style of processing that allows us to apply handlers to nodes in the tree as they are being converted to R objects. This is a style of event-driven or asynchronous calling

In addition to the generic token event handlers such as "begin an XML element" (the startElement handler), one can also provide handler functions for specific tags/elements such as <myTag> with handler elements with the same name as the XML element of interest, i.e. "myTag" = function(x, attrs).

When the event parser is reading text nodes, it may call the text handler function with different sub-strings of the text within the node. Essentially, the parser collects up n characters into a buffer and passes this as a single string the text handler and then continues collecting more text until the buffer is full or there is no more text. It passes each sub-string to the text handler. If trim is TRUE, it removes leading and trailing white space from the substring before calling the text handler. If the resulting text is empty and ignoreBlanks is TRUE, then we don't bother calling the text handler function.

So the key thing to remember about dealing with text is that the entire text of a node may come in multiple separate calls to the text handler. A common idiom is to have the text handler concatenate the values it is passed in separate calls and to have the end element handler process the entire text and reset the text variable to be empty.

Usage

```
xmlEventParse(file, handlers = xmlEventHandler(),
              ignoreBlanks = FALSE, addContext=TRUE,
               useTagName = TRUE, asText = FALSE, trim=TRUE,
                useExpat=FALSE, isURL = FALSE,
                 state = NULL, replaceEntities = TRUE, validate = FALSE,
                  saxVersion = 1, branches = NULL,
                   useDotNames = length(grep("^\\.", names(handlers))) > 0,
                    error = xmlErrorCumulator(), addFinalizer = NA,
                     encoding = character())
```

Arguments

file	the source of the XML content. This can be a string giving the name of a file or remote URL, the XML itself, a connection object, or a function. If this is a string, and asText is TRUE, the value is the XML content. This allows one to read the content separately from parsing without having to write it to a file. If asText is FALSE and a string is passed for file, this is taken as the name of a file or remote URI. If one is using the libxml parser (i.e. not expat), this can be a URI accessed via HTTP or FTP or a compressed local file. If it is the name of a local file, it can include ~, environment variables, etc. which will be expanded by R. (Note this is not the case in S-Plus, as far as I know.) If a connection is given, the parser incrementally reads one line at a time by calling the function readLines with the connection as the first argument (and 1 as the number of lines to read). The parser calls this function each time it needs more input. If invoking the readLines function to get each line is excessively slow or is inappropriate, one can provide a function as the value of fileName. Again, when the XML parser needs more content to process, it invokes this function to get a string. This function is called with a single argument, the maximum size of the string that can be returned. The function is responsible for accessing the correct connection(s), etc. which is typically done via lexical scoping/environments. This mechanism allows the user to control how the XML content is retrieved in very general ways. For example, one might read from a set of files, starting one when the contents of the previous file have been consumed. This allows for the use of hybrid connection objects. Support for connections and functions in this form is only provided if one is using libxml2 and not libxml version 1.
handlers	a closure object that contains functions which will be invoked as the XML components in the document are encountered by the parser. The standard function or handler names are startElement(), endElement() comment(), getEntity, entityDeclaration(), processingInstruction(), text(), cdata(), startDocument(), and endDocument(), or alternatively and preferrably, these names prefixed with a '.', i.e. .startElement, .comment, ... The call signature for the entityDeclaration function was changed in version 1.7-0. Note that in earlier versions, the C routine did not invoke any R function and so no code will actually break. Also, we have renamed externalEntity to getEntity. These were based on the expat parser.

The new signature is `c(name = "character", type = "integer",content = "", system = "character", public = "character" )` `name` gives the name of the entity being defined. The `type` identifies the type of the entity using the value of a C-level enumerated constant used in libxml2, but also gives the human-readable form as the name of the single element in the integer vector. The possible values are `"Internal_General"`, `"External_General_Parsed"`, `"External_General_Unparsed"`, `"Internal_Parameter"`, `"External_Parameter"`, `"Internal_Predefined"`.

If we are dealing with an internal entity, the content will be the string containing the value of the entity. If we are dealing with an external entity, then `content` will be a character vector of length 0, i.e. empty. Instead, either or both of the system and public arguments will be non-empty and identify the location of the external content. `system` will be a string containing a URI, if non-empty, and `public` corresponds to the PUBLIC identifier used to identify content using an SGML-like approach. The use of PUBLIC identifiers is less common.

`ignoreBlanks` a logical value indicating whether text elements made up entirely of white space should be included in the resulting 'tree'.

`addContext` logical value indicating whether the callback functions in 'handlers' should be invoked with contextual information about the parser and the position in the tree, such as node depth, path indices for the node relative the root, etc. If this is True, each callback function should support ....

`useTagName` a logical value. If this is `TRUE`, when the SAX parser signals an event for the start of an XML element, it will first look for an element in the list of handler functions whose name matches (exactly) the name of the XML element. If such an element is found, that function is invoked. Otherwise, the generic `startElement` handler function is invoked. The benefit of this is that the author of the handler functions can write node-specific handlers for the different element names in a document and not have to establish a mechanism to invoke these functions within the `startElement` function. This is done by the XML package directly.

If the value is `FALSE`, then the `startElement` handler function will be called without any effort to find a node-specific handler. If there are no node-specific handlers, specifying `FALSE` for this parameter will make the computations very slightly faster.

`asText` logical value indicating that the first argument, 'file', should be treated as the XML text to parse, not the name of a file. This allows the contents of documents to be retrieved from different sources (e.g. HTTP servers, XML-RPC, etc.) and still use this parser.

`trim` whether to strip white space from the beginning and end of text strings.

`useExpat` a logical value indicating whether to use the expat SAX parser, or to default to the libxml. If this is TRUE, the library must have been compiled with support for expat. See supportsExpat.

`isURL` indicates whether the `file` argument refers to a URL (accessible via ftp or http) or a regular file on the system. If `asText` is TRUE, this should not be specified.

`state` an optional S object that is passed to the callbacks and can be modified to communicate state between the callbacks. If this is given, the callbacks should accept an argument named `.state` and it should return an object that will be used as the

updated value of this state object. The new value can be any S object and will be passed to the next callback where again it will be updated by that functions return value, and so on. If this not specified in the call to `xmlEventParse`, no `.state` argument is passed to the callbacks. This makes the interface compatible with previous releases.

`replaceEntities`

logical value indicating whether to substitute entity references with their text directly. This should be left as False. The text still appears as the value of the node, but there is more information about its source, allowing the parse to be reversed with full reference information.

`saxVersion` an integer value which should be either 1 or 2. This specifies which SAX interface to use in the C code. The essential difference is the number of arguments passed to the `startElement` handler function(s). Under SAX 2, in addition to the name of the element and the named-attributes vector, two additional arguments are provided. The first identifies the namespace of the element. This is a named character vector of length 1, with the value being the URI of the namespace and the name being the prefix that identifies that namespace within the document. For example, `xmlns:r="http://www.r-project.org"` would be passed as `c(r = "http://www.r-project.org")`. If there is no prefix because the namespace is being used as the default, the result of calling `names` on the string is `""`. The second additional argument (the fourth in total) gives the collection of all the namespaces defined within this element. Again, this is a named character vector.

`validate` Currently, this has no effect as the libxml2 parser uses a document structure to do validation. a logical indicating whether to use a validating parser or not, or in other words check the contents against the DTD specification. If this is true, warning messages will be displayed about errors in the DTD and/or document, but the parsing will proceed except for the presence of terminal errors.

`branches` a named list of functions. Each element identifies an XML element name. If an XML element of that name is encountered in the SAX stream, the stream is processed until the end of that element and an internal node (see `xmlTreeParse` and its `useInternalNodes` parameter) is created. The function in our branches list corresponding to this XML element is then invoked with the (internal) node as the only argument. This allows one to use the DOM model on a sub-tree of the entire document and thus use both SAX and DOM together to get the efficiency of SAX and the simpler programming model of DOM.

Note that the branches mechanism works top-down and does not work for nested tags. If one specifies an element name in the `branches` argument, e.g. myNode, and there is a nested myNode instance within a branch, the branches handler will not be called for that nested instance. If there is an instance where this is problematic, please contact the maintainer of this package.

One can cause the parser to collect a branch without identifying the node within the `branches` list. Specifically, within a regular start-element handler, one can return a function whose class is `SAXBranchFunction`. The SAX parser recognizes this and collects up the branch starting at the current node being processed and when it is complete, invokes this function. This allows us to dynamically determine which nodes to treat as branches rather than just matching names.

This is necessary when a node name has different meanings in different parts of the XML hierarchy, e.g. dict in an iTunes song list.

See the file `itunesSax2.R` inthe examples for an example of this.

This is a two step process. In the future, we might make it so that the R function handling the start-element event could directly collect the branch and continue its operations without having to call another function asynchronously.

`useDotNames` a logical value indicating whether to use the newer format for identifying general element function handlers with the '.' prefix, e.g. .text, .comment, .startElement. If this is `FALSE`, then the older format text, comment, startElement, ... are used. This causes problems when there are indeed nodes named text or comment or startElement as a node-specific handler are confused with the corresponding general handler of the same name. Using `TRUE` means that your list of handlers should have names that use the '.' prefix for these general element handlers. This is the preferred way to write new code.

`error` a function that is called when an XML error is encountered. This is called with 6 arguments and is described in `xmlTreeParse`.

`addFinalizer` a logical value or identifier for a C routine that controls whether we register finalizers on the intenal node.

`encoding` a character string (scalar) giving the encoding for the document. This is optional as the document should contain its own encoding information. However, if it doesn't, the caller can specify this for the parser.

Details

This is now implemented using the libxml parser. Originally, this was implemented via the Expat XML parser by Jim Clark (`http://www.jclark.com/`).

Value

The return value is the 'handlers' argument. It is assumed that this is a closure and that the callback functions have manipulated variables local to it and that the caller knows how to extract this.

Note

The libxml parser can read URLs via http or ftp. It does not require the support of `wget` as used in other parts of R, but uses its own facilities to connect to remote servers.

The idea for the hybrid SAX/DOM mode where we consume tokens in the stream to create an entire node for a sub-tree of the document was first suggested to me by Seth Falcon at the Fred Hutchinson Cancer Research Center. It is similar to the XML::Twig module in Perl by Michel Rodriguez.

Author(s)

Duncan Temple Lang

References

`https://www.w3.org/XML/`, `http://www.jclark.com/xml/`

See Also

xmlTreeParse xmlStopParser XMLParserContextFunction

Examples

```
 fileName <- system.file("exampleData", "mtcars.xml", package="XML")

   # Print the name of each XML tag encountered at the beginning of each
   # tag.
   # Uses the libxml SAX parser.
 xmlEventParse(fileName,
                list(startElement=function(name, attrs){
                                cat(name,"\n")
                              }),
                useTagName=FALSE, addContext = FALSE)


## Not run:
  # Parse the text rather than a file or URL by reading the URL's contents
  # and making it a single string. Then call xmlEventParse
xmlURL <- "https://www.omegahat.net/Scripts/Data/mtcars.xml"
xmlText <- paste(scan(xmlURL, what="",sep="\n"),"\n",collapse="\n")
xmlEventParse(xmlText, asText=TRUE)

## End(Not run)

    # Using a state object to share mutable data across callbacks
f <- system.file("exampleData", "gnumeric.xml", package = "XML")
zz <- xmlEventParse(f,
                    handlers = list(startElement=function(name, atts, .state) {
                                                    .state = .state + 1
                                                    print(.state)
                                                    .state
                                                }), state = 0)
print(zz)



    # Illustrate the startDocument and endDocument handlers.
xmlEventParse(fileName,
               handlers = list(startDocument = function() {
                                                  cat("Starting document\n")
                                               },
                               endDocument = function() {
                                                  cat("ending document\n")
                                             }),
               saxVersion = 2)
```

```
if(libxmlVersion()$major >= 2) {


 startElement = function(x, ...) cat(x, "\n")

 xmlEventParse(ff <- file(f), handlers = list(startElement = startElement))

 # Parse with a function providing the input as needed.
 xmlConnection =
  function(con) {

   if(is.character(con))
     con = file(con, "r")

   if(isOpen(con, "r"))
     open(con, "r")

   function(len) {

     if(len < 0) {
        close(con)
        return(character(0))
     }

      x = character(0)
      tmp = ""
    while(length(tmp) > 0 && nchar(tmp) == 0) {
      tmp = readLines(con, 1)
      if(length(tmp) == 0)
        break
      if(nchar(tmp) == 0)
        x = append(x, "\n")
      else
        x = tmp
    }
    if(length(tmp) == 0)
      return(tmp)

    x = paste(x, collapse="")

    x
  }
 }

 ## this leaves a connection open
 ## xmlConnection would need amending to return the connection.
 ff = xmlConnection(f)
 xmlEventParse(ff, handlers = list(startElement = startElement))


  # Parse from a connection. Each time the parser needs more input, it
  # calls readLines(<con>, 1)
 xmlEventParse(ff <-file(f),  handlers = list(startElement = startElement))
```

```
 # using SAX 2
h = list(startElement = function(name, attrs, namespace, allNamespaces){
                                cat("Starting", name,"\n")
                                if(length(attrs))
                                    print(attrs)
                                print(namespace)
                                print(allNamespaces)
                            },
         endElement = function(name, uri) {
                             cat("Finishing", name, "\n")
           })
xmlEventParse(system.file("exampleData", "namespaces.xml", package="XML"),
              handlers = h, saxVersion = 2)


# This example is not very realistic but illustrates how to use the
# branches argument. It forces the creation of complete nodes for
# elements named <b> and extracts the id attribute.
# This could be done directly on the startElement, but this just
# illustrates the mechanism.
filename = system.file("exampleData", "branch.xml", package="XML")
b.counter = function() {
               nodes <- character()
               f = function(node) { nodes <<- c(nodes, xmlGetAttr(node, "id"))}
               list(b = f, nodes = function() nodes)
            }

 b = b.counter()
 invisible(xmlEventParse(filename, branches = b["b"]))
 b$nodes()


 filename = system.file("exampleData", "branch.xml", package="XML")

 invisible(xmlEventParse(filename, branches = list(b = function(node) {
                         print(names(node))})))
 invisible(xmlEventParse(filename, branches = list(b = function(node) {
                         print(xmlName(xmlChildren(node)[[1]]))})))
}


 ##########################################
 # Stopping the parser mid-way and an example of using XMLParserContextFunction.

 startElement =
 function(ctxt, name, attrs, ...)  {
   print(ctxt)
     print(name)
     if(name == "rewriteURI") {
          cat("Terminating parser\n")
   xmlStopParser(ctxt)
     }
```

```
}
class(startElement) = "XMLParserContextFunction"
endElement =
function(name, ...)
  cat("ending", name, "\n")

fileName = system.file("exampleData", "catalog.xml", package = "XML")
xmlEventParse(fileName, handlers = list(startElement = startElement,
                                        endElement = endElement))
```

`xmlGetAttr` *Get the value of an attribute in an XML node*

Description

This is a convenience function that retrieves the value of a named attribute in an XML node, taking care of checking for its existence. It also allows the caller to provide a default value to use as the return value if the attribute is not present.

Usage

```
xmlGetAttr(node, name, default = NULL, converter = NULL,
            namespaceDefinition = character(),
             addNamespace = length(grep(":", name)) > 0)
```

Arguments

`node`	the XML node
`name`	the name of the attribute
`default`	a value to use as the default return if the attribute is not present in the XML node.
`converter`	an optional function which if supplied is invoked with the attribute value and the value returned. This can be used to convert the string to an arbitrary value which is useful if it is, for example, a number. This is only called if the attribute exists within the node. In other words, it is not applied to the `default` value.
`namespaceDefinition`	a named character vector giving name space prefixes and URIs to use when resolving for the the attribute with a namespace. The values are used to compare the name space prefix used in the `name` given by the user to the name space definition in the node to ensure they match. This is important as we might ask for an attribute named `r:width` assuming that the prefix `r` corresponded to the URI `http://www.r-project.org`. However, there may be a name space prefix `r` defined on the node that points to a different URI and so this would be an erroneous match.
`addNamespace`	a logical value that indicates whether we should put the namespace prefix on the resulting name. This is passed on to `xmlAttrs` and so controls whether the resulting attribute names have the prefix attached. So one specifies `TRUE` for this argument if the attribute identifier has a namespace prefix.

Details

This just checks that the attribute list is non-NULL and that there is an element with the specified name.

Value

If the attribute is present, the return value is a string which is the value of the attribute. Otherwise, the value of `default` is returned.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlAttrs

Examples

```
node <- xmlNode("foo", attrs=c(a="1", b="my name"))

xmlGetAttr(node, "a")
xmlGetAttr(node, "doesn't exist", "My own default value")

xmlGetAttr(node, "b", "Just in case")
```

---

xmlHandler *Example XML Event Parser Handler Functions*

Description

A closure containing simple functions for the different types of events potentially called by the xmlEventParse, and some tag-specific functions to illustrate how one can add functions for specific DTDs and XML element types. Contains a local list which can be mutated by invocations of the closure's function.

Usage

```
xmlHandler()
```

Value

List containing the functions enumerated in the closure definition along with the list.

Note

This is just an example.

Author(s)

Duncan Temple Lang

See Also

xmlEventParse, xmlTreeParse

Examples

```
## Not run:
xmlURL <- "https://www.omegahat.net/Scripts/Data/mtcars.xml"
xmlText <- paste(scan(xmlURL, what="", sep="\n"),"\n",collapse="\n")

## End(Not run)

xmlURL <- system.file("exampleData", "mtcars.xml", package="XML")
xmlText <- paste(readLines(xmlURL), "\n", collapse="")
xmlEventParse(xmlText, handlers = NULL, asText=TRUE)
xmlEventParse(xmlText, xmlHandler(), useTagName=TRUE, asText=TRUE)
```

---

xmlHashTree *Constructors for trees stored as flat list of nodes with information about parents and children.*

Description

These (and related internal) functions allow us to represent trees as a simple, non-hierarchical collection of nodes along with corresponding tables that identify the parent and child relationships. This is different from representing a tree as a list of lists of lists ... in which each node has a list of its own children. In a functional language like R, it is not possible then for the children to be able to identify their parents.

We use an environment to represent these flat trees. Since these are mutable without requiring the change to be reassigned, we can modify a part of the tree locally without having to reassign the top-level object.

We can use either a list (with names) to store the nodes or a hash table/associative array that uses names. There is a non-trivial performance difference.

Usage

```
xmlHashTree(nodes = list(), parents = character(), children = list(),
            env = new.env(TRUE, parent = emptyenv()))
```

Arguments

nodes	a collection of existing nodes that are to be added to the tree. These are used to initialize the tree. If this is specified, you must also specify children and parents.
parents	the parent relationships for the nodes given by nodes.
children	the children relationships for the nodes given by nodes.
env	an environment in which the information for the tree will be stored. This is essentially the tree object as it allows us to modify parts of the tree without having to reassign the top-level object. Unlike most R data types, environments are mutable.

Value

An xmlHashTree object has an accessor method via $ for accessing individual nodes within the tree. One can use the node name/identifier in an expression such as tt$myNode to obtain the element. The name of a node is either its XML node name or if that is already present in the tree, a machine generated name.

One can find the names of all the nodes using the objects function since these trees are regular environments in R. Using the all = TRUE argument, one can also find the "hidden" elements that make define the tree's structure. These are .children and .parents. The former is an (hashed) environment. Each element is identified by the node in the tree by the node's identifier (corresponding to the name of the node in the tree's environment). The value of that element is simply a character vector giving the identifiers of all of the children of that node.

The .parents element is also an environemnt. Each element in this gives the pair of node and parent identifiers with the parent identifier being the value of the variable in the environment. In other words, we look up the parent of a node named 'kid' by retrieving the value of the variable 'kid' in the .parents environment of this hash tree.

The function .addNode is used to insert a new node into the tree.

The structure of this tree allows one to easily travers all nodes, navigate up the tree from a node via its parent. Certain tasks are more complex as the hierarchy is not implicit within a node.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

xmlTreeParse xmlTree xmlOutputBuffer xmlOutputDOM

Examples

```
f = system.file("exampleData", "dataframe.xml", package = "XML")
tr  = xmlHashTree()
xmlTreeParse(f, handlers = list(.startElement = tr[[".addNode"]]))

tr # print the tree on the screen

 # Get the two child nodes of the dataframe node.
xmlChildren(tr$dataframe)

 # Find the names of all the nodes.
objects(tr)
 # Which nodes have children
objects(tr$.children)

 # Which nodes are leaves, i.e. do not have children
setdiff(objects(tr), objects(tr$.children))

 # find the class of each of these leaf nodes.
sapply(setdiff(objects(tr), objects(tr$.children)),
        function(id) class(tr[[id]]))

 # distribution of number of children
sapply(tr$.children, length)


 # Get the first A node
tr$A

 # Get is parent node.
xmlParent(tr$A)


f = system.file("exampleData", "allNodeTypes.xml", package = "XML")

  # Convert the document
r = xmlInternalTreeParse(f, xinclude = TRUE)
ht = as(r, "XMLHashTree")
ht

 # work on the root node, or any node actually
as(xmlRoot(r), "XMLHashTree")

 # Example of making copies of an XMLHashTreeNode object to create a separate tree.
f = system.file("exampleData", "simple.xml", package = "XML")
tt = as(xmlParse(f), "XMLHashTree")

xmlRoot(tt)[[1]]
xmlRoot(tt)[[1, copy = TRUE]]

table(unlist(eapply(tt, xmlName)))
# if any of the nodes had any attributes
```

```
# table(unlist(eapply(tt, xmlAttrs)))
```

XMLInternalDocument-class

Class to represent reference to C-level data structure for an XML document

---

Description

This class is used to provide a handle/reference to a C-level data structure that contains the information from parsing parsing XML content. This leaves the nodes in the DOM or tree as C-level nodes rather than converting them to explicit R XMLNode objects. One can then operate on this tree in much the same way as one can the XMLNode representations, but we a) avoid copying the nodes to R, and b) can navigate the tree both down and up using xmlParent giving greater flexibility. Most importantly, one can use an XMLInternalDocument class object with an XPath expression to easily and relatively efficiently find nodes within a document that satisfy some criterion. See getNodeSet.

Objects from the Class

Objects of this type are created via xmlTreeParse and htmlTreeParse with the argument useInternalNodes given as TRUE.

Extends

Class oldClass, directly.

Methods

There are methods to serialize (dump) a document to a file or as a string, and to coerce it to a node by finding the top-level node of the document. There are functions to search the document for nodes specified by an XPath expression.

References

XPath https://www.w3.org/TR/xpath/

See Also

xmlTreeParse htmlTreeParse getNodeSet

Examples

```
f = system.file("exampleData", "mtcars.xml", package="XML")
doc = xmlParse(f)
getNodeSet(doc, "//variables[@count]")
getNodeSet(doc, "//record")

getNodeSet(doc, "//record[@id='Mazda RX4']")

# free(doc)
```

xmlName *Extraces the tag name of an XMLNode object.*

Description

Each XMLNode object has an element or tag name introduced in the <name . . .> entry in an XML document. This function returns that name.

We can also set that name using `xmlName(node) <- "name"` and the value can have an XML name space prefix, e.g. `"r:name"`.

Usage

```
xmlName(node, full = FALSE)
xmlName(x) <- value
```

Arguments

node, x	The XMLNode object whose tag name is being requested.
full	a logical value indicating whether to prepend the namespace prefix, if there is one, or return just the name of the XML element/node. TRUE means prepend the prefix.
value	New tag name for x.

Value

A character vector of length 1 which is the node$name entry.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlChildren, xmlAttrs, xmlTreeParse

Examples

```
fileName <- system.file("exampleData", "test.xml", package="XML")
doc <- xmlTreeParse(fileName)
xmlName(xmlRoot(doc)[[1]])

tt = xmlRoot(doc)[[1]]
xmlName(tt)
xmlName(tt) <- "bob"
```

```
 # We can set the node on an internal object also.
n = newXMLNode("x")

xmlName(n)
xmlName(n) <- "y"


xmlName(n) <- "r:y"
```

`xmlNamespace` *Retrieve the namespace value of an XML node.*

Description

Each XML node has a namespace identifier which is a string indicating in which DTD (Document Type Definition) the definition of that element can be found. This avoids the problem of having different document definitions using the same names for XML elements that have different meaning. To resolve the name space, i.e. i.e. find out to where the identifier points, one can use the expression `xmlNamespace(xmlRoot(doc))`.

The class of the result is is an S3-style object of class `XMLNamespace`.

Usage

```
xmlNamespace(x)
xmlNamespace(x, ...) <- value
```

Arguments

`x`	the object whose namespace is to be computed
`value`	the prefix for a namespace that is defined in the node or any of the ancestors.
`...`	additional arguments for setting the name space

Value

For non-root nodes, this returns a string giving the identifier of the name space for this node. For the root node, this returns a list with 2 elements:

`id`	the identifier by which other nodes refer to this namespace.
`uri`	the URI or location that defines this namespace.
`local`	? (can't remember off-hand).

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlName xmlChildren xmlAttrs xmlValue xmlNamespaceDefinitions

Examples

```
doc <- xmlTreeParse(system.file("exampleData", "job.xml", package="XML"))
xmlNamespace(xmlRoot(doc))
xmlNamespace(xmlRoot(doc)[[1]][[1]])


doc <- xmlInternalTreeParse(system.file("exampleData", "job.xml", package="XML"))
   # Since the first node, xmlRoot() will skip that, by default.
xmlNamespace(xmlRoot(doc))
xmlNamespace(xmlRoot(doc)[[1]][[1]])

node <- xmlNode("arg", xmlNode("name", "foo"), namespace="R")
xmlNamespace(node)


doc = xmlParse('<top xmlns:r="http://www.r-project.org"><bob><code>a = 1:10</code></bob></top>')
 node = xmlRoot(doc)[[1]][[1]]
 xmlNamespace(node) = "r"
 node


doc = xmlParse('<top xmlns:r="http://www.r-project.org"><bob><code>a = 1:10</code></bob></top>')
 node = xmlRoot(doc)[[1]][[1]]
 xmlNamespaces(node, set = TRUE) = c(omg = "https://www.omegahat.net")
 node
```

`xmlNamespaceDefinitions` *Get definitions of any namespaces defined in this XML node*

Description

If the given node has any namespace definitions declared within it, i.e. of the form `xmlns:myNamespace="http://www.myNS.`
`xmlNamespaceDefinitions` provides access to these definitions. While they appear in the XML node in the document as attributes, they are treated differently by the parser and so do not show up in the nodes attributes via `xmlAttrs`.

`getDefaultNamespace` is used to get the default namespace for the top-level node in a document.

The `recursive` parameter allows one to conveniently find all the namespace definitions in a document or sub-tree without having to examine the file. This can be useful when working with XPath queries via `getNodeSet`.

Usage

```
xmlNamespaceDefinitions(x, addNames = TRUE, recursive = FALSE, simplify = FALSE, ...)
xmlNamespaces(x, addNames = TRUE, recursive = FALSE,  simplify = FALSE, ...)
getDefaultNamespace(doc, ns = xmlNamespaceDefinitions(doc, simplify = simplify),
                    simplify = FALSE)
```

Arguments

x	the XMLNode object in which to find any namespace definitions
addNames	a logical indicating whether to compute the names for the elements in the resulting list. The names are convenient, but one can avoid the (very small) overhead of computing these with this parameter.
doc	the XMLInternalDocument object obtained from a call to xmlParse
recursive	a logical value indicating whether to extract the namespace definitions for just this node (FALSE) or all of the descendant nodes as well (TRUE). If this is TRUE, all the namespace definitions are collected into a single "flat" list and so there may be duplicate names.
simplify	a logical value. If this is TRUE, a character vector of prefix-URI pairs is returned. This can be used directly in calls to functions such as xpathApply and getNodeSet. The default value of FALSE returns a list of name space definitions which also identify whether the definition is local to the particular node or inherited from an ancestor.
ns	the collection of namespaces. This is typically omitted but can be specified if it has been computed in an earlier step.
...	additional parameters for methods

Value

A list with as many elements as there are namespace definitions. Each element is an object of class XMLNameSpace, containing fields giving the local identifier, the associated defining URI and a logical value indicating whether the definition is local to this node. The name of each element is the prefix or alias used for that namespace definition, i.e. the value of the id field in the namespace definition. For default namespaces, i.e. those that have no prefix/alias, the name is "".

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

xmlTreeParse xmlAttrs xmlGetAttr

Examples

```
f = system.file("exampleData", "longitudinalData.xml", package = "XML")
n = xmlRoot(xmlTreeParse(f))
xmlNamespaceDefinitions(n)
xmlNamespaceDefinitions(n, recursive = TRUE)


  # Now using internal nodes.
f = system.file("exampleData", "namespaces.xml", package = "XML")
doc = xmlInternalTreeParse(f)
n = xmlRoot(doc)
xmlNamespaceDefinitions(n)

xmlNamespaceDefinitions(n, recursive = TRUE)
```

---

xmlNode *Create an XML node*

Description

These functions allow one to create XML nodes as are created in C code when reading XML documents. Trees of XML nodes can be constructed and integrated with other trees generated manually or with via the parser.

Usage

```
xmlNode(name, ..., attrs=NULL, namespace="", namespaceDefinitions = NULL,
        .children = list(...))
xmlTextNode(value, namespace="", entities = XMLEntities, cdata = FALSE)
xmlPINode(sys, value, namespace="")
xmlCDataNode(...)
xmlCommentNode(text)
```

Arguments

name	The tag or element name of the XML node. This is what appears in the elements as <name> .. </name>
...	The children nodes of this XML node. These can be objects of class XMLNode or arbitrary values that will be converted to a string to form an XMLTextNode object.
.children	an alternative mechanism to specifying the children which is useful for programmatic use when one has the children in an existing list. The ... mechanism is for use when the children are specified directly and individually.
attrs	A named character vector giving the name, value pairs of attributes for this XML node.
value	This is the text that is to be used when forming an XMLTextNode.

cdata — a logical value which controls whether the text being used for the child node is to be first enclosed within a CDATA node to escape special characters such as > and &.

namespace — The XML namespace identifier for this node.

namespaceDefinitions — a collection of name space definitions, containing the prefixes and the corresponding URIs. This is most conveniently specified as a character vector whose names attribute is the vector of prefixes and whose values are the URIs. Alternatively, one can provide a list of name space definition objects such as those returned

sys — the name of the system for which the processing instruction is targeted. This is the value that appears in the <?sys value?>

text — character string giving the contents of the comment.

entities — a character vector giving the mapping from special characters to their entity equivalent. This provides the character-expanded entity pairings of 'character = entity' , e.g. '<' = "lt" which are used to make the content valid XML so that it can be used within a text node. The text searched sequentially for instances of each character in the names and each instance is replaced with the corresponding '&entity;'

Value

An object of class XMLNode. In the case of xmlTextNode, this also inherits from XMLTextNode. The fields or slots that objects of these classes have include name, attributes, children and namespace. However, one should the accessor functions xmlName, xmlAttrs, xmlChildren and xmlNamespace

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

addChildren xmlTreeParse asXMLNode newXMLNode newXMLPINode newXMLCDataNode newXMLCommentNode

Examples

```
# node named arg with two children: name and defaultValue
# Both of these have a text node as their child.
n <- xmlNode("arg", attrs = c(default="TRUE"),
              xmlNode("name", "foo"), xmlNode("defaultValue","1:10"))


 # internal C-level node.
a = newXMLNode("arg", attrs = c(default = "TRUE"),
```

```
                newXMLNode("name", "foo"),
                newXMLNode("defaultValue", "1:10"))

xmlAttrs(a) = c(a = 1, b = "a string")

xmlAttrs(a) = c(a = 1, b = "a string", append = FALSE)

newXMLNamespace(a, c("r" = "http://www.r-project.org"))
xmlAttrs(a) = c("r:class" = "character")


xmlAttrs(a[[1]]) = c("r:class" = "character")


 # Using a character vector as a namespace definitions
x = xmlNode("bob",
            namespaceDefinitions = c(r = "http://www.r-project.org",
                                     omg = "https://www.omegahat.net"))
```

XMLNode-class *Classes to describe an XML node object.*

Description

These classes are intended to represent an XML node, either directly in S or a reference to an internal libxml node. Such nodes respond to queries about their name, attributes, namespaces and children. These are old-style, S3 class definitions at present.

Slots

These are old-style S3 class definitions and do not have formal slots

Methods

No methods defined with class "XMLNode" in the signature.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.xmlsoft.org

See Also

xmlTreeParse xmlTree newXMLNode xmlNode

Examples

```
          # An R-level XMLNode object
 a <- xmlNode("arg", attrs = c(default="T"),
              xmlNode("name", "foo"), xmlNode("defaultValue","1:10"))

xmlAttrs(a) = c(a = 1, b = "a string")
```

`xmlOutputBuffer` *XML output streams*

Description

These two functions provide different ways to construct XML documents incrementally. They provide a single, common interface for adding and closing tags, and inserting nodes. The buffer version stores the XML representation as a string. The DOM version builds the tree of XML node objects entirely within R.

Usage

```
xmlOutputBuffer(dtd=NULL, nameSpace="", buf=NULL,
                            nsURI=NULL, header="<?xml version=\"1.0\"?>")

xmlOutputDOM(tag="doc", attrs = NULL, dtd=NULL,
                         nameSpace=NULL, nsURI=character(0),
                         xmlDeclaration = NULL)
```

Arguments

`dtd`	a DTD object (see `parseDTD` and `xmlTreeParse`) which contains specifications about what elements are valid within other elements and what attributes are supported by different elements. This can be used to validate the document as it is being constructed incrementally.
`attrs`	attributes for the top-level node, in the form of a named vector or list.
`nameSpace`	the default namespace identifier to be used when an element is created without an explicit namespace. This provides a convenient way to specify the default name space that appers in tags throughout the resulting document.
`buf`	a connection object or a string into which the XML content is written. This is currently a simplistic implementation since we will use the OOP-style classes from the Omegahat projects in the future.
`nsURI`	the URI or value for the name space which is used when declaring the namespace. For `xmlOuputDOM`, this is a named character vector with each element giving the name space identifier and the corresponding URI, \ e.g `c(shelp = "https://www.omegahat.net/XML/SHelp")`
`header`	if non-NULL, this is immediately written to the output stream allowing one to control the initial section of the XML document.

tag	the name of the top-level node/element in the DOM being created.
xmlDeclaration	a logical value or a string. If this is a logical value and TRUE, the default <?xml version='1.0'?> processing instruction is emitted at the top of the document. If it is FALSE, no xml declaration is emitted at the top of the document. If this is provided as a string, the contents of this is added as the content of the processing instruction. A version='1.0' is added if there is no 'version=' content within the given string.

Details

These functions create a closure instance which provides methods or functions that operate on shared data used to represent the contents of the XML document being created and the current state of that creation.

Value

Both of these functions return a list of functions which operate on the XML data in a shared environment.

value	get the contents of the XML document as they are currently defined.
addTag	add a new element to the document, specifying its name and attributes. This allows the tag to be left open so that new elements will be added as children of it.
closeTag	close the currently open tag, indicating that new elements will be added, by default, as siblings of this one.
reset	discard the current contents of the document so that we can start over and free the resources (memory) associated with this document.

The following are specific to xmlOutputDOM:

addNode	insert an complete XMLNode object into the currently active (i.e. open) node.
current	obtain the path or collection of indices to to the currently active/open node from the root node.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

xmlTree for a native/internal (C-level) representation of the tree, xmlNode, xmlTextNode, append.xmlNode

And a different representation of a tree is available via xmlHashTree.

Examples

```
 con <- xmlOutputDOM()
con$addTag("author", "Duncan Temple Lang")
con$addTag("address",  close=FALSE)
 con$addTag("office", "2C-259")
 con$addTag("street", "Mountain Avenue.")
 con$addTag("phone", close = FALSE)
   con$addTag("area", "908", attrs=c(state="NJ"))
   con$addTag("number", "582-3217")
 con$closeTag() # phone
con$closeTag() # address

con$addTag("section", close = FALSE)
 con$addNode(xmlTextNode("This is some text "))
 con$addTag("a","and a link", attrs=c(href="https://www.omegahat.net"))
 con$addNode(xmlTextNode("and some follow up text"))

 con$addTag("subsection", close = FALSE)
   con$addNode(xmlTextNode("some addtional text "))
   con$addTag("a", attrs=c(href="https://www.omegahat.net"), close=FALSE)
     con$addNode(xmlTextNode("the content of the link"))
   con$closeTag() # a
 con$closeTag() # "subsection"
con$closeTag() # section


 d <- xmlOutputDOM()
 d$addPI("S", "plot(1:10)")
 d$addCData('x <- list(1, a="&");\nx[[2]]')
 d$addComment("A comment")
 print(d$value())
 print(d$value(), indent = FALSE, tagSeparator = "")


 d = xmlOutputDOM("bob", xmlDeclaration = TRUE)
 print(d$value())

 d = xmlOutputDOM("bob", xmlDeclaration = "encoding='UTF-8'")
 print(d$value())


 d = xmlOutputBuffer("bob", header = "<?xml version='1.0' encoding='UTF-8'?>",
                     dtd = "foo.dtd")
 d$addTag("bob")
 cat(d$value())
```

`xmlParent` *Get parent node of XMLInternalNode or ancestor nodes*

Description

xmlParent operates on an XML node and returns a reference to its parent node within the document tree. This works for an internal, C-level XMLInternalNode object created, for examply, using newXMLNode and related functions or xmlTree or from xmlTreeParse with the useInternalNodes parameter.

It is possible to find the parent of an R-level XML node when using a tree created with, for example, xmlHashTree as the parent information is stored separately.

xmlAncestors walks the chain of parens to the top of the document and either returns a list of those nodes, or alternatively a list of the values obtained by applying a function to each of the nodes.

Usage

```
xmlParent(x, ...)
xmlAncestors(x, fun = NULL, ..., addFinalizer = NA, count = -1L)
```

Arguments

x	an object of class XMLInternalNode whose parent is being requested.
fun	an R function which is invoked for each node as we walk up the tree.
...	any additional arguments that are passed in calls to fun after the node object and for xmlParent this allows methods to define their own additional parameters.
addFinalizer	a logical value indicating whether the default finalizer routine should be registered to free the internal xmlDoc when R no longer has a reference to this external pointer object. This can also be the name of a C routine or a reference to a C routine retrieved using getNativeSymbolInfo.
count	an integer that indicates how many levels of the hierarchy to traverse. This allows us to get the count most recent ancestors of the node.

Details

This uses the internal libxml structures to access the parent in the DOM tree. This function is generic so that we can add methods for other types of nodes if we so want in the future.

Value

xmlParent returns object of class XMLInternalNode.

If fun is NULL, xmlAncestors returns a list of the nodes in order of top-most node or root of the tree, then its child, then the child of that child, etc. This is the reverse order in which the nodes are visited/found.

If fun is a function, xmlAncestors returns a list whose elements are the results of calling that function for each node. Again, the order is top down.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/

See Also

xmlChildren xmlTreeParse xmlNode

Examples

```
top = newXMLNode("doc")
s = newXMLNode("section", attr = c(title = "Introduction"))
a = newXMLNode("article", s)
addChildren(top, a)

xmlName(xmlParent(s))
xmlName(xmlParent(xmlParent(s)))


  # Find the root node.
root = a
while(!is.null(xmlParent(root)))
    root = xmlParent(root)

 # find the names of the parent nodes of each 'h' node.
 # use a global variable to "simplify" things and not use a closure.

filename = system.file("exampleData", "branch.xml", package = "XML")
parentNames <- character()
xmlParse(filename,
              handlers =
                list(h = function(x) {
                 parentNames <<- c(parentNames, xmlName(xmlParent(x)))
                }))

table(parentNames)
```

---

xmlParseDoc *Parse an XML document with options controlling the parser.*

Description

This function is a generalization of xmlParse that parses an XML document. With this function, we can specify a combination of different options that control the operation of the parser. The options control many different aspects the parsing process

Usage

```
xmlParseDoc(file, options = 1L, encoding = character(),
            asText = !file.exists(file), baseURL = file)
```

Arguments

file	the name of the file or URL or the XML content itself
options	options controlling the behavior of the parser. One specifies the different options as elements of an integer vector. These are then bitwised OR'ed together. The possible options are RECOVER, NOENT, DTDLOAD, DTDATTR, DTDVALID, NOERROR, NOWARNING, PEDANTIC, NOBLANKS, SAX1, XINCLUDE, NONET, NODICT, NSCLEAN, NOCDATA, NOXINNODE, COMPACT, OLD10, NOBASEFIX, HUGE, OLDSAX. (These options are also listed in the (non-exported) variable parserOptions.)
encoding	character string that provides the encoding of the document if it is not explicitly contained within the document itself.
asText	a logical value indicating whether file is the XML content (TRUE) or the name of a file or URL (FALSE)
baseURL	the base URL used for resolving relative documents, e.g. XIncludes. This is important if file is the actual XML content rather than a URL

Value

An object of class XMLInternalDocument.

Author(s)

Duncan Temple Lang

References

libxml2

See Also

xmlParse

Examples

```
f = system.file("exampleData", "mtcars.xml", package="XML")
    # Same as xmlParse()
xmlParseDoc(f)

txt =
    '<top xmlns:r="http://www.r-project.org">
       <b xmlns:r="http://www.r-project.org">
         <c xmlns:omg="http:/www.omegahat.net"/>
       </b>
     </top>'

xmlParseDoc(txt,  NSCLEAN, asText = TRUE)

txt =
    '<top xmlns:r="http://www.r-project.org"  xmlns:r="http://www.r-project.org">
       <b xmlns:r="http://www.r-project.org">
```

```
          <c xmlns:omg="http:/www.omegahat.net"/>
        </b>
      </top>'

xmlParseDoc(txt, c(NSCLEAN, NOERROR), asText = TRUE)
```

`xmlParserContextFunction` *Identifies function as expecting an xmlParserContext argument*

Description

This is a convenience function for setting the class of the specified function to include `"XMLParserContextFunction"`. This identifies it as expecting an `xmlParserCtxt` object as its first argument. The resulting function can be passed to the internal/native XML parser as a handler/callback function. When the parser calls it, it recognizes this class information and includes a reference to the C-level `xmlParserCtxt` object as the first argument in the call.

This `xmlParserCtxt` object can be used to gracefully terminate the parsing (without an error), and in the future will also provide access to details about the current state of the parser, e.g. the encoding of the file, the XML version, whether entities are being replaced, line and column number for each node processed.

Usage

```
xmlParserContextFunction(f, class = "XMLParserContextFunction")
```

Arguments

`f`	the function whose class information is to be augmented.
`class`	the name of the class which is to be added to the `class` attribute of the function.

Value

The function object `f` whose class attribute has been prepended with the value of `class`.

Author(s)

Duncan Temple Lang

See Also

`xmlInternalTreeParse`/`xmlParse` and the `branches` parameter of `xmlEventParse`.

Examples

```
fun = function(context, ...) {
           # do things to parse the node
           # using the context if necessary.
         cat("In XMLParserContextFunction\n")
         xmlStopParser(context)
       }
fun = xmlParserContextFunction(fun)

txt = "<doc><a/></doc>"
# doesn't work for xmlTreeParse()
#   xmlTreeParse(txt, handlers = list(a = fun))

#  but does in xmlEventParse().
xmlEventParse(txt, handlers = list(startElement = fun), asText = TRUE)
```

`xmlRoot` *Get the top-level XML node.*

Description

These are a collection of methods for providing easy access to the top-level `XMLNode` object resulting from parsing an XML document. They simplify accessing this node in the presence of auxillary information such as DTDs, file name and version information that is returned as part of the parsing.

Usage

```
xmlRoot(x, skip = TRUE, ...)
## S3 method for class 'XMLDocumentContent'
xmlRoot(x, skip = TRUE, ...)
## S3 method for class 'XMLInternalDocument'
xmlRoot(x, skip = TRUE, addFinalizer = NA, ...)
## S3 method for class 'HTMLDocument'
xmlRoot(x, skip = TRUE, ...)
```

Arguments

`x`	the object whose root/top-level XML node is to be returned.
`skip`	a logical value that controls whether DTD nodes and/or XMLComment objects that appear before the "real" top-level node of the document should be ignored (`TRUE`) or not (`FALSE`) when returning the root node.
`...`	arguments that are passed by the generic to the different specialized methods of this generic.
`addFinalizer`	a logical value or identifier for a C routine that controls whether we register finalizers on the intenal node.

Value

An object of class XMLNode.

Note

One cannot obtain the parent or top-level node of an XMLNode object in S. This is different from languages like C, Java, Perl, etc. and is primarily because S does not provide support for references.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlTreeParse [[.XMLNode

Examples

```
doc <- xmlTreeParse(system.file("exampleData", "mtcars.xml", package="XML"))
xmlRoot(doc)
 # Note that we cannot use getSibling () on a regular R-level XMLNode object
 # since we cannot go back up or across the tree from that node, but
 # only down to the children.

  # Using an internal node via xmlParse (== xmlInternalTreeParse())
doc <- xmlParse(system.file("exampleData", "mtcars.xml", package="XML"))
n = xmlRoot(doc, skip = FALSE)
   # skip over the DTD and the comment
d = getSibling(getSibling(n))
```

---

xmlSchemaValidate *Validate an XML document relative to an XML schema*

Description

This function validates an XML document relative to an XML schema to ensure that it has the correct structure, i.e. valid sub-nodes, attributes, etc.

The xmlSchemaValidationErrorHandler is a function that returns a list of functions which can be used to cumulate or collect the errors and warnings from the schema validation operation.

Usage

```
xmlSchemaValidate(schema, doc,
                                  errorHandler = xmlErrorFun(),
                                   options = 0L)

schemaValidationErrorHandler()
```

Arguments

schema	an object of class xmlSchemaRef which is usually the result of a call to xmlInternalTreeParse with isSchema = TRUE, or xmlSchemaParse.
doc	an XML document which has already been parsed into a XMLInternalDocument or which is a file name or string which is coerced to an XMLInternalDocument-class object
options	an integer giving the options controlling the validation. At present, this is either 0 or 1 and is essentially irrelevant to us. It may be of value in the future.
errorHandler	a function or a list whose first element is a function which is then used as the collector for the warning and error messages reported during the validation. For each warning or error, this function is invoked and the class of the message is either XMLSchemaWarning or XMLSchemaError respectively.

Value

Typically, a list with 3 elements:

status	0 for validated, and non-zero for invalid
errors	a character vector
warnings	a character vector

If an empty error handler is provided (i.e. NULL) just an integer indicating the status of the validation is returned. 0 indicates everything was okay; a non-zero value indicates a validation error. (-1 indicates an internal error in libxml2)

References

libxml2 www.xmlsoft.org

See Also

xmlSchemaParse

Examples

```
if(FALSE) {
 xsd = xmlParse(system.file("exampleData", "author.xsd", package = "XML"), isSchema =TRUE)
  doc = xmlInternalTreeParse(system.file("exampleData", "author.xml",  package = "XML"))
  xmlSchemaValidate(xsd, doc)
}
```

`xmlSearchNs` *Find a namespace definition object by searching ancestor nodes*

Description

This function allows one to search an XML tree from a particular node and find the namespace definition for a given namespace prefix or URL. This namespace definition can then be used to set it on a node to make it the effective namespace for that node.

Usage

```
xmlSearchNs(node, ns, asPrefix = TRUE, doc = as(node, "XMLInternalDocument"))
```

Arguments

`node`	an `XMLInternaElementNode`
`ns`	a character string (vector of length 1). If `asPrefix` is `TRUE`, this is the namespace alias/prefix. If `asPrefix` is `FALSE`, this is the URL of the namespace definition
`asPrefix`	a logical value. See `ns`.
`doc`	the XML document in which the node(s) are located

Value

An object of class XMLNamespaceRef.

Author(s)

Duncan Temple Lang

References

libxml2

See Also

`newXMLNode`

Examples

```
txt = '<top xmlns:r="http://www.r-project.org"><section><bottom/></section></top>'

doc = xmlParse(txt)
bottom = xmlRoot(doc)[[1]][[1]]
xmlSearchNs(bottom, "r")
```

xmlSerializeHook *Functions that help serialize and deserialize XML internal objects*

Description

These functions can be used to control how the C-level data structures associated with XML documents, nodes, XPath queries, etc. are serialized to a a file or connection and deserialized back into an R session. Since these C-level data structures are represented in R as external pointers, they would normally be serialized and deserialized in a way that loses all the information about the contents of the memory being referenced. `xmlSerializeHook` arranges to serialize these pointers by saving the corresponding XML content as a string and also the class of the object. The deserialize function converts such objects back to their original form.

These functions are used in calls to `saveRDS` and `readRDS` via the `refhook` argument. `saveRDS(obj, filename, refhook = xmlSerializeHook) readRDS(filename, refhook = xmlDeserializeHook)`

Usage

```
xmlSerializeHook(x)
xmlDeserializeHook(x)
```

Arguments

x	the object to be deserialized, and the character vector to be deserialized.

Value

`xmlSerializeHook` returns a character version of the XML document or node, along with the basic class. If it is called with an object that is not an native/internal XML object, it returns `NULL`

`xmlDeserializeHook` returns the parsed XML object, either a document or a node.

Author(s)

Duncan Temple Lang

References

The R Internals Manual.

See Also

`saveRDS` and `readRDS`

Examples

```
z = newXMLNode("foo")
f = system.file("exampleData", "tides.xml", package = "XML")
doc = xmlParse(f)
hdoc = as(doc, "XMLHashTree")

nodes = getNodeSet(doc, "//pred")

ff <- file.path(tempdir(), "tmp.rda")
saveRDS(list(a = 1:10, z = z, doc = doc, hdoc = hdoc, nodes = nodes), ff,
          refhook = xmlSerializeHook)

v = readRDS(ff, refhook = xmlDeserializeHook)
unlink(ff)
```

---

xmlSize *The number of sub-elements within an XML node.*

Description

XML elements can contain other, nested sub-elements. This generic function determines the number of such elements within a specified node. It applies to an object of class XMLNode or XMLDocument.

Usage

```
xmlSize(obj)
## Default S3 method:
xmlSize(obj)
```

Arguments

obj	An an object of class XMLNode or XMLDocument.

Value

an integer which is the length of the value from xmlChildren.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlChildren, xmlAttrs, xmlName, xmlTreeParse

Examples

```
fileName <- system.file("exampleData", "mtcars.xml", package="XML")
doc <- xmlTreeParse(fileName)
xmlSize(doc)
xmlSize(doc$doc$children[["dataset"]][["variables"]])
```

`xmlSource` *Source the R code, examples, etc. from an XML document*

Description

This is the equivalent of a smart `source` for extracting the R code elements from an XML document and evaluating them. This allows for a "simple" way to collect R functions definitions or a sequence of (annotated) R code segments in an XML document along with other material such as notes, documentation, data, FAQ entries, etc., and still be able to access the R code directly from within an R session. The approach enables one to use the XML document as a container for a heterogeneous collection of related material, some of which is R code. In the literate programming parlance, this function essentially dynamically "tangles" the document within R, but can work on small subsets of it that are easily specified in the `xmlSource` function call. This is a convenient way to annotate code in a rich way and work with source files in a new and potentially more effective manner.

`xmlSourceFunctions` provides a convenient way to read only the function definitions, i.e. the `<r:function>` nodes. We can restrict to a subset by specifying the node ids of interest.

`xmlSourceSection` allows us to evaluate the code in one or more specific sections.

This style of authoring code supports mixed language support in which we put, for example, C and R code together in the same document. Indeed, one can use the document to store arbitrary content and still retrieve the R code. The more structure there is, the easier it is to create tools to extract that information using XPath expressions.

We can identify individual `r:code` nodes in the document to process, i.e. evaluate. We do this using their `id` attribute and specifying which to process via the `ids` argument. Alternatively, if a document has a node `r:codeIds` as a child of the top-level node (or within an invisible node), we read its contents as a sequence of line separated `id` values as if they had been specified via the argument `ids` to this function.

We can also use XSL to extract the code. See `getCode.xsl` in the Omegahat XSL collection.

This particular version (as opposed to other implementations) uses XPath to conveniently find the nodes of interest.

Usage

```
xmlSource(url, ...,
          envir = globalenv(),
          xpath = character(),
          ids = character(),
          omit = character(),
          ask = FALSE,
          example = NA,
```

```
            fatal = TRUE, verbose = TRUE, echo = verbose, print = echo,
            xnodes = DefaultXMLSourceXPath,
            namespaces = DefaultXPathNamespaces, section = character(),
            eval = TRUE, init = TRUE, setNodeNames = FALSE, parse = TRUE,
            force = FALSE)
    xmlSourceFunctions(doc, ids = character(), parse = TRUE, ...)
    xmlSourceSection(doc, ids = character(),
                     xnodes = c(".//r:function", ".//r:init[not(@eval='false')]",
                                ".//r:code[not(@eval='false')]",
                                ".//r:plot[not(@eval='false')]"),
                     namespaces = DefaultXPathNamespaces, ...)
```

Arguments

url	the name of the file, URL containing the XML document, or an XML string. This is passed to xmlTreeParse which is called with useInternalNodes = TRUE.
...	additional arguments passed to xmlTreeParse
envir	the environment in which the code elements of the XML document are to be evaluated. By default, they are evaluated in the global environment so that assignments take place there.
xpath	a string giving an XPath expression which is used after parsing the document to filter the document to a particular subset of nodes. This allows one to restrict the evaluation to a subset of the original document. One can do this directly by parsing the XML document, applying the XPath query and then passing the resulting node set to this xmlSource function's appropriate method. This argument merely allows for a more convenient form of those steps, collapsing it into one action.
ids	a character vector. XML nodes containing R code (e.g. r:code, r:init, r:function, r:plot) can have an id attribute. This vector allows the caller to specify the subset of these nodes to process, i.e. whose code will be evaluated. The order is currently not important. It may be used in the future to specify the order in which the nodes are evaluated. If this is not specified and the document has a node r:codeIds as an immediate child of the top-most node, the contents of this node or contained within an invisible node (so that it doesn't have to be filtered when rendering the document), the names of the r:code id values to process are taken as the individual lines from the body of this node.
omit	a character vector. The values of the id attributes of the nodes that we want to skip or omit from the evaluation. This allows us to specify the set that we don't want evaluated, in contrast to the ids argument. The order is not important.
ask	logical
example	a character or numeric vector specifying the values of the id attributes of any r:example nodes in the document. A single document may contain numerous, separate examples and these can be marked uniquely using an id attribute, e.g. <r:example id=''. This argument allows the caller to specify which example (or examples) to run. If this is not specified by the caller and there are r:example

nodes in the document, the user is prompted to select an example via a (text-based) menu. If a character vector is given by the caller, we use partial matching against the collection of `id` attributes of the r:example nodes to identify the examples of interest. Alternatively, one can specify the example(s) to run by number.

`fatal` (currently unused) a logical value. The idea is to control how we handle errors when evaluating individual code segments. We could recover from errors and continue processing subsequent nodes.

`verbose` a logical value. If `TRUE`, information about what code segments are being evaluated is displayed on the console. `echo` controls whether code is displayed, but this controls whether additional informatin is also displayed. See `source`.

`xnodes` a character vector. This is a collection of xpath expressions given as individual strings which find the nodes whose contents we evaluate.

`echo` a logical value indicating whether to display the code before it is evaluated.

`namespaces` a named character vector (i.e. name = value pairs of strings) giving the prefix - URI pairings for the namespaces used in the XPath expressions. The URIs must match those in the document, but the prefixes are local to the XPath expression. The default provides mappings for the prefixes "r", "omg", "perl", "py", and so on. See `XML:::DefaultXPathNamespaces`.

`section` a vector of numbers or strings. This allows the caller to specify that the function should only look for R-related nodes within the specified section(s). This is useful for being able to easily process only the code in a particular subset of the document identified by a DocBook `section` node. A string value is used to match the `id` attribute of the `section` node. A number (assumed to be an integer) is used to index the set of `section` nodes. These amount to XPath expressions of the form `//section[number]` and `//section[@id = string]`.

`print` a logical value indicating whether to print the results

`eval` a logical value indicating whether to evaluate the code in the specified nodes or to just return the result of parsing the text in each node.

`init` a logical controlling whether to run the R code in any r:init nodes.

`doc` the XML document, either a file name, the content of the document or the parsed document.

`parse` a logical value that controls whether we parse the code or just return the text representation from the XML without parsing it. This allows us to get just the code.

`setNodeNames` a logical value that controls whether we compute the name for each node (or result) by finding is id or name attribute or enclosing task node.

`force` a logical value. If this is `TRUE`, the function will evaluate the code in a node even if it is explicitly marked as not to be evaluated with `eval = "false"`, either on the node itself or an ancestor.

Details

This evaluates the `code`, `function` and `example` elements in the XML content that have the appropriate namespace (i.e. r, s, or no namespace) and discards all others. It also discards r:output nodes

from the text, along with processing instructions and comments. And it resolves `r:frag` or `r:code` nodes with a `ref` attribute by identifying the corresponding `r:code` node with the same value for its `id` attribute and then evaluating that node in place of the `r:frag` reference.

Value

An R object (typically a list) that contains the results of evaluating the content of the different selected code segments in the XML document. We use `sapply` to iterate over the nodes and so If the results of all the nodes A list giving the pairs of expressions and evaluated objects for each of the different XML elements processed.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

`xmlTreeParse`

Examples

```
xmlSource(system.file("exampleData", "Rsource.xml", package="XML"))

 # This illustrates using r:frag nodes.
 # The r:frag nodes are not processed directly, but only
 # if referenced in the contents/body of a r:code node
f = system.file("exampleData", "Rref.xml", package="XML")
xmlSource(f)
```

`xmlStopParser` *Terminate an XML parser*

Description

This function allows an R-level function to terminate an XML parser before it completes the processing of the XML content. This might be useful, for example, in event-driven parsing with `xmlEventParse` when we want to read through an XML file until we find a record of interest. Then, having retrieved the necessary information, we want to terminate the parsing rather than let it pointlessly continue. Instead of raising an error in our handler function, we can call `xmlStopParser` and return. The parser will then take control again and terminate and return back to the original R function from which it was invoked.

The only argument to this function is a reference to internal C-level which identifies the parser. This is passed by the R-XML parser mechanism to a function invoked by the parser if that function inherits (in the S3 sense) from the class `XMLParserContextFunction`.

Usage

```
xmlStopParser(parser)
```

Arguments

parser	an object of class XMLParserContext which must have been obtained by via an XMLParserContextFunction function called by the parser. This is just a handler function whose class includes XMLParserContextFunction

Value

TRUE if it succeeded and an error is raised if the parser object is not valid.

Author(s)

Duncan Temple Lang

References

libxml2 http://xmlsoft.org

See Also

xmlEventParse

Examples

```
##########################################
# Stopping the parser mid-way and an example of using XMLParserContextFunction.

startElement =
function(ctxt, name, attrs, ...)  {
  print(ctxt)
    print(name)
    if(name == "rewriteURI") {
         cat("Terminating parser\n")
  xmlStopParser(ctxt)
    }
}
class(startElement) = "XMLParserContextFunction"
endElement =
function(name, ...)
  cat("ending", name, "\n")

fileName = system.file("exampleData", "catalog.xml", package = "XML")
xmlEventParse(fileName, handlers = list(startElement = startElement, endElement = endElement))
```

xmlStructuredStop *Condition/error handler functions for XML parsing*

Description

These functions provide basic error handling for the XML parser in R. They also illustrate the basics which will allow others to provide customized error handlers that make more use of the information provided in each error reported.

The `xmlStructuredStop` function provides a simple R-level handler for errors raised by the XML parser. It collects the information provided by the XML parser and raises an R error. This is only used if `NULL` is specified for the `error` argument of `xmlTreeParse`, `xmlTreeParse` and `htmlTreeParse`.

The default is to use the function returned by a call to `xmlErrorCumulator` as the error handler. This, as the name suggests, cumulates errors. The idea is to catch each error and let the parser continue and then report them all. As each error is encountered, it is collected by the function. If `immediate` is `TRUE`, the error is also reported on the console. When the parsing is complete and has failed, this function is invoked again with a zero-length character vector as the message (first argument) and then it raises an error. This function will then raise an R condition of class `class`.

Usage

```
xmlStructuredStop(msg, code, domain, line, col, level, filename,
                  class = "XMLError")
xmlErrorCumulator(class = "XMLParserErrorList", immediate = TRUE)
```

Arguments

`msg`	character string, the text of the message being reported
`code`	an integer code giving an identifier for the error (see xmlerror.h) for the moment,
`domain`	an integer domain indicating in which "module" or part of the parsing the error occurred, e.g. name space, parser, tree, xinclude, etc.
`line`	an integer giving the line number in the XML content being processed corresponding to the error,
`col`	an integer giving the column position of the error,
`level`	an integer giving the severity of the error ranging from 1 to 3 in increasing severity (warning, error, fatal),
`filename`	character string, the name of the document being processed, i.e. its file name or URL.
`class`	character vector, any classes to prepend to the class attribute to make the error/condition. These are prepended to those returned via `simpleError`.
`immediate`	logical value, if `TRUE` errors are displayed on the R console as they are encountered. Otherwise, the errors are collected and displayed at the end of the XML parsing.

Value

This calls stop and so does not return a value.

Author(s)

Duncan Temple Lang

References

libxml2 and its error handling facilities (http://xmlsoft.org

See Also

xmlTreeParse xmlInternalTreeParse htmlTreeParse

Examples

```
tryCatch( xmlTreeParse("<a><b></a>", asText = TRUE, error = NULL),
               XMLError = function(e) {
                  cat("There was an error in the XML at line",
                        e$line, "column", e$col, "\n",
                       e$message, "\n")
              })
```

---

xmlToDataFrame *Extract data from a simple XML document*

Description

This function can be used to extract data from an XML document (or sub-document) that has a simple, shallow structure that does appear reasonably commonly. The idea is that there is a collection of nodes which have the same fields (or a subset of common fields) which contain primitive values, i.e. numbers, strings, etc. Each node corresponds to an "observation" and each of its sub-elements correspond to a variable. This function then builds the corresponding data frame, using the union of the variables in the different observation nodes. This can handle the case where the nodes do not all have all of the variables.

Usage

```
xmlToDataFrame(doc, colClasses = NULL, homogeneous = NA,
               collectNames = TRUE, nodes = list(),
               stringsAsFactors = FALSE)
```

Arguments

doc	the XML content. This can be the name of a file containing the XML, the parsed XML document. If one wants to work on a subset of nodes, specify these via the nodes parameter.
colClasses	a list/vector giving the names of the R types for the corresponding variables and this is used to coerce the resulting column in the data frame to this type. These can be named. This is similar to the colClasses parameter for read.table. If this is given as a list, columns in the data frame corresponding to elements that are NULL are omitted from the answer. This can be slightly complex to specify if the different nodes have the "variables" in quite different order as there is not a well defined order for the variables corresponding to colClasses.
homogeneous	a logical value that indicates whether each of the nodes contains all of the variables (TRUE) or if there may be some nodes which have only a subset of them. The function determines this if the caller does not specify homogeneous or uses NA as the value. It is a parameter to allow the caller to specify this information and avoid these "extra" computations. If the caller knows this information it is more efficient to specify it.
collectNames	a logical value indicating whether we compute the names by explicitly computing the union of all variable names or, if FALSE, we use the names from the node with the most children. This latter case is useful when the caller knows that the there is at least one node with all the variables.
nodes	a list of XML nodes which are to be processed
stringsAsFactors	a logical value that controls whether character vectors are converted to factor objects in the resulting data frame.

Value

A data frame.

Author(s)

Duncan Temple Lang

See Also

xmlParse getNodeSet

Examples

```
f = system.file("exampleData", "size.xml", package = "XML")
xmlToDataFrame(f, c("integer", "integer", "numeric"))

  # Drop the middle variable.
z = xmlToDataFrame(f, colClasses = list("integer", NULL, "numeric"))


  #  This illustrates how we can get a subset of nodes and process
```

```
 #  those as the "data nodes", ignoring the others.
f = system.file("exampleData", "tides.xml", package = "XML")
doc = xmlParse(f)
xmlToDataFrame(nodes = xmlChildren(xmlRoot(doc)[["data"]]))

  # or, alternatively
xmlToDataFrame(nodes = getNodeSet(doc, "//data/item"))


f = system.file("exampleData", "kiva_lender.xml", package = "XML")
doc = xmlParse(f)
dd = xmlToDataFrame(getNodeSet(doc, "//lender"))
```

`xmlToList` *Convert an XML node/document to a more R-like list*

Description

This function is an early and simple approach to converting an XML node or document into a more typical R list containing the data values directly (rather than as XML nodes). It is useful for dealing with data that is returned from REST requests or other Web queries or generally when parsing XML and wanting to be able to access the content as elements in a list indexed by the name of the node. For example, if given a node of the form `<x> <a>text</a> <b foo="1"/> <c bar="me"> <d>a phrase</d> </c> </x>` We would end up with a list with elements named "a", "b" and "c". "a" would be the string "text", b would contain the named character vector `c(foo = "1")` (i.e. the attributes) and "c" would contain the list with two elements named "d" and ".attrs". The element corresponding to "d" is a character vector with the single element "a phrase". The ".attrs" element of the list is the character vector of attributes from the node `<c>...</c>`.

Usage

```
xmlToList(node, addAttributes = TRUE, simplify = FALSE)
```

Arguments

`node`	the XML node or document to be converted to an R list. This can be an "internal" or C-level node (i.e. `XMLInternalNode-class`) or a regular R-level node (either `XMLNode-class` or `XMLHashNode`).
`addAttributes`	a logical value which controls whether the attributes of an empty node are added to the
`simplify`	a logical value that controls whether we collapse the list to a vector if the elements all have a common compatible type. Basically, this controls whether we use `sapply` or `lapply`.

Value

A list whose elements correspond to the children of the top-level nodes.

Author(s)

Duncan Temple Lang

See Also

xmlTreeParse getNodeSet and xpathApply xmlRoot, xmlChildren, xmlApply, [[, etc. for accessing the content of XML nodes.

Examples

```
tt =
 '<x>
    <a>text</a>
    <b foo="1"/>
    <c bar="me">
       <d>a phrase</d>
    </c>
  </x>'

  doc = xmlParse(tt)
  xmlToList(doc)

   # use an R-level node representation
  doc = xmlTreeParse(tt)
  xmlToList(doc)
```

---

xmlToS4 *General mechanism for mapping an XML node to an S4 object*

Description

This generic function and its methods recursively process an XML node and its child nodes (and theirs and so on) to map the nodes to S4 objects.

This is the run-time function that corresponds to the makeClassTemplate function.

Usage

```
xmlToS4(node, obj = new(xmlName(node)), ...)
```

Arguments

node	the top-level XML node to convert to an S4 object
obj	the object whose slots are to be filled from the information in the XML node
...	additional parameters for methods

Value

The object obj whose slots have been modified.

Author(s)

Duncan Temple Lang

See Also

makeClassTemplate

Examples

```
txt = paste0("<doc><part><name>ABC</name><type>XYZ</type>',
             <cost>3.54</cost><status>available</status></part></doc>")
doc = xmlParse(txt)

setClass("part", representation(name = "character",
                                type = "character",
                                cost = "numeric",
                                status= "character"))
xmlToS4(xmlRoot(doc)[["part"]])
```

---

xmlTree *An internal, updatable DOM object for building XML trees*

Description

This is a mutable object (implemented via a closure) for representing an XML tree, in the same spirit as xmlOutputBuffer and xmlOutputDOM but that uses the internal structures of libxml. This can be used to create a DOM that can be constructed in R and exported to another system such as XSLT.

Usage

```
xmlTree(tag, attrs = NULL, dtd=NULL, namespaces=list(),
          doc = newXMLDoc(dtd, namespaces))
```

Arguments

tag	the node or element name to use to create the new top-level node in the tree or alternatively, an XMLInternalNode that was already created. This is optional. If it is not specified, no top-most node is created but can be added using addNode. If a top-level tag is added in the call to xmlTree, that becomes the currently active or open node (e.g. same as addNode(..., close = FALSE)) and nodes subsequently added to this
attrs	attributes for the top-level node, in the form of a named character vector.
dtd	the name of the external DTD for this document. If specified, this adds the DOCTYPE node to the resulting document. This can be a node created earlier with a call to newXMLDTDNode, or alternatively it can be a character vector with 1, 2 or 3 elements giving the name of the top-level node, and the public identifier and the system identifier for the DTD in that order.

namespaces — a named character vector with each element giving the name space identifier and the corresponding URI, \e.g `c(shelp = "https://www.omegahat.net/XML/SHelp")` If `tag` is specified as a character vector, these name spaces are defined within that new node.

doc — an internal XML document object, typically created with `newXMLDoc`. This is used as the host document for all the new nodes that will be created as part of this document. If one wants to create nodes without an internal document ancestor, one can alternatively specify this is as `NULL`.

Details

This creates a collection of functions that manipulate a shared state to build and maintain an XML tree in C-level code.

Value

An object of class `XMLInternalDOM` that extends `XMLOutputStream` and has the same interface (i.e. "methods") as `xmlOutputBuffer` and `xmlOutputDOM`. Each object has methods for adding a new XML tag, closing a tag, adding an XML comment, and retrieving the contents of the tree.

addTag — create a new tag at the current position, optionally leaving it as the active open tag to which new nodes will be added as children

closeTag — close the currently active tag making its parent the active element into which new nodes will be added.

addComment — add an XML comment node as a child of the active node in the document.

value — retrieve an object representing the XML tree. See `saveXML` to serialize the contents of the tree.

add — degenerate method in this context.

Note

This is an early version of this function and I need to iron out some of the minor details.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.xmlsoft.org

See Also

saveXML newXMLDoc newXMLNode xmlOutputBuffer xmlOutputDOM

Examples

```
z = xmlTree("people", namespaces = list(r = "http://www.r-project.org"))
z$setNamespace("r")

z$addNode("person", attrs = c(id = "123"), close = FALSE)
  z$addNode("firstname", "Duncan")
  z$addNode("surname", "Temple Lang")
  z$addNode("title", "Associate Professor")
  z$addNode("expertize", close = FALSE)
     z$addNode("topic", "Data Technologies")
     z$addNode("topic", "Programming Language Design")
     z$addNode("topic", "Parallel Computing")
     z$addNode("topic", "Data Visualization")
     z$addNode("topic", "Meta-Computing")
     z$addNode("topic", "Inter-system interfaces")
  z$closeTag()
  z$addNode("address", "4210 Mathematical Sciences Building, UC Davis")
z$closeTag()

  tr <- xmlTree("CDataTest")
  tr$addTag("top", close=FALSE)
  tr$addCData("x <- list(1, a='&');\nx[[2]]")
  tr$addPI("S", "plot(1:10)")
  tr$closeTag()
  cat(saveXML(tr$value()))


  f = tempfile()
  saveXML(tr, f, encoding = "UTF-8")


  # Creating a node
x = rnorm(3)
z = xmlTree("r:data", namespaces = c(r = "http://www.r-project.org"))
z$addNode("numeric", attrs = c("r:length" = length(x)))


  # shows namespace prefix on an attribute, and different from the one on the node.
  z = xmlTree()
z$addNode("r:data",  namespace = c(r = "http://www.r-project.org",
                                   omg = "https://www.omegahat.net"),
         close = FALSE)
x = rnorm(3)
z$addNode("r:numeric", attrs = c("omg:length" = length(x)))


z = xmlTree("examples")
z$addNode("example", namespace = list(r = "http://www.r-project.org"), close = FALSE)
z$addNode("code", "mean(rnorm(100))", namespace = "r")


x = summary(rnorm(1000))
```

```
d = xmlTree()
d$addNode("table", close = FALSE)

d$addNode("tr", .children = sapply(names(x), function(x) d$addNode("th", x)))
d$addNode("tr", .children = sapply(x, function(x) d$addNode("td", format(x))))

d$closeNode()
cat(saveXML(d))

# Dealing with DTDs and system and public identifiers for DTDs.
# Just doctype
za = xmlTree("people", dtd = "people")
### www.omegahat.net is flaky
# no public element
zb = xmlTree("people",
             dtd = c("people", "", "https://www.omegahat.net/XML/types.dtd"))
# public and system
zc = xmlTree("people",
             dtd = c("people", "//a//b//c//d",
                     "https://www.omegahat.net/XML/types.dtd"))
```

`xmlTreeParse` *XML Parser*

Description

Parses an XML or HTML file or string containing XML/HTML content, and generates an R structure representing the XML/HTML tree. Use `htmlTreeParse` when the content is known to be (potentially malformed) HTML. This function has numerous parameters/options and operates quite differently based on their values. It can create trees in R or using internal C-level nodes, both of which are useful in different contexts. It can perform conversion of the nodes into R objects using caller-specified handler functions and this can be used to map the XML document directly into R data structures, by-passing the conversion to an R-level tree which would then be processed recursively or with multiple descents to extract the information of interest.

`xmlParse` and `htmlParse` are equivalent to the `xmlTreeParse` and `htmlTreeParse` respectively, except they both use a default value for the `useInternalNodes` parameter of `TRUE`, i.e. they working with and return internal nodes/C-level nodes. These can then be searched using XPath expressions via `xpathApply` and `getNodeSet`.

`xmlSchemaParse` is a convenience function for parsing an XML schema.

Usage

```
xmlTreeParse(file, ignoreBlanks=TRUE, handlers=NULL, replaceEntities=FALSE,
             asText=FALSE, trim=TRUE, validate=FALSE, getDTD=TRUE,
             isURL=FALSE, asTree = FALSE, addAttributeNamespaces = FALSE,
             useInternalNodes = FALSE, isSchema = FALSE,
             fullNamespaceInfo = FALSE, encoding = character(),
             useDotNames = length(grep("^\\.", names(handlers))) > 0,
```

```
             xinclude = TRUE, addFinalizer = TRUE, error = xmlErrorCumulator(),
             isHTML = FALSE, options = integer(), parentFirst = FALSE)

xmlInternalTreeParse(file, ignoreBlanks=TRUE, handlers=NULL, replaceEntities=FALSE,
             asText=FALSE, trim=TRUE, validate=FALSE, getDTD=TRUE,
             isURL=FALSE, asTree = FALSE, addAttributeNamespaces = FALSE,
             useInternalNodes = TRUE, isSchema = FALSE,
             fullNamespaceInfo = FALSE, encoding = character(),
             useDotNames = length(grep("^\\.", names(handlers))) > 0,
             xinclude = TRUE, addFinalizer = TRUE, error = xmlErrorCumulator(),
             isHTML = FALSE, options = integer(), parentFirst = FALSE)

xmlNativeTreeParse(file, ignoreBlanks=TRUE, handlers=NULL, replaceEntities=FALSE,
             asText=FALSE, trim=TRUE, validate=FALSE, getDTD=TRUE,
             isURL=FALSE, asTree = FALSE, addAttributeNamespaces = FALSE,
             useInternalNodes = TRUE, isSchema = FALSE,
             fullNamespaceInfo = FALSE, encoding = character(),
             useDotNames = length(grep("^\\.", names(handlers))) > 0,
             xinclude = TRUE, addFinalizer = TRUE, error = xmlErrorCumulator(),
             isHTML = FALSE, options = integer(), parentFirst = FALSE)


htmlTreeParse(file, ignoreBlanks=TRUE, handlers=NULL, replaceEntities=FALSE,
             asText=FALSE, trim=TRUE, validate=FALSE, getDTD=TRUE,
             isURL=FALSE, asTree = FALSE, addAttributeNamespaces = FALSE,
             useInternalNodes = FALSE, isSchema = FALSE,
             fullNamespaceInfo = FALSE, encoding = character(),
             useDotNames = length(grep("^\\.", names(handlers))) > 0,
             xinclude = TRUE, addFinalizer = TRUE, error = htmlErrorHandler,
             isHTML = TRUE, options = integer(), parentFirst = FALSE)

htmlParse(file, ignoreBlanks = TRUE, handlers = NULL, replaceEntities = FALSE,
          asText = FALSE, trim = TRUE, validate = FALSE, getDTD = TRUE,
           isURL = FALSE, asTree = FALSE, addAttributeNamespaces = FALSE,
           useInternalNodes = TRUE, isSchema = FALSE, fullNamespaceInfo = FALSE,
             encoding = character(),
             useDotNames = length(grep("^\\.", names(handlers))) > 0,
              xinclude = TRUE, addFinalizer = TRUE,
               error = htmlErrorHandler, isHTML = TRUE,
                options = integer(), parentFirst = FALSE)

xmlSchemaParse(file, asText = FALSE, xinclude = TRUE, error = xmlErrorCumulator())

xmlParse(file, ignoreBlanks = TRUE, handlers = NULL, replaceEntities = FALSE,
         asText = FALSE, trim = TRUE, validate = FALSE, getDTD = TRUE,
         isURL = FALSE, asTree = FALSE, addAttributeNamespaces = FALSE,
         useInternalNodes = TRUE, isSchema = FALSE, fullNamespaceInfo = FALSE,
         encoding = character(),
```

```
          useDotNames = length(grep("^\\.",  names(handlers))) > 0,
          xinclude = TRUE, addFinalizer = TRUE, error = xmlErrorCumulator(),
          isHTML = FALSE, options = integer(), parentFirst = FALSE)
```

Arguments

file	The name of the file containing the XML contents. This can contain ~ which is expanded to the user's home directory. It can also be a URL. See isURL. Additionally, the file can be compressed (gzip) and is read directly without the user having to de-compress (gunzip) it.
ignoreBlanks	logical value indicating whether text elements made up entirely of white space should be included in the resulting 'tree'.
handlers	Optional collection of functions used to map the different XML nodes to R objects. Typically, this is a named list of functions, and a closure can be used to provide local data. This provides a way of filtering the tree as it is being created in R, adding or removing nodes, and generally processing them as they are constructed in the C code. In a recent addition to the package (version 0.99-8), if this is specified as a single function object, we call that function for each node (of any type) in the underlying DOM tree. It is invoked with the new node and its parent node. This applies to regular nodes and also comments, processing instructions, CDATA nodes, etc. So this function must be sufficiently general to handle them all.
replaceEntities	logical value indicating whether to substitute entity references with their text directly. This should be left as False. The text still appears as the value of the node, but there is more information about its source, allowing the parse to be reversed with full reference information.
asText	logical value indicating that the first argument, file, should be treated as the XML text to parse, not the name of a file. This allows the contents of documents to be retrieved from different sources (e.g. HTTP servers, XML-RPC, etc.) and still use this parser.
trim	whether to strip white space from the beginning and end of text strings.
validate	logical indicating whether to use a validating parser or not, or in other words check the contents against the DTD specification. If this is true, warning messages will be displayed about errors in the DTD and/or document, but the parsing will proceed except for the presence of terminal errors. This is ignored when parsing an HTML document.
getDTD	logical flag indicating whether the DTD (both internal and external) should be returned along with the document nodes. This changes the return type. This is ignored when parsing an HTML document.
isURL	indicates whether the file argument refers to a URL (accessible via ftp or http) or a regular file on the system. If asText is TRUE, this should not be specified. The function attempts to determine whether the data source is a URL by using grep to look for http or ftp at the start of the string. The libxml parser handles the connection to servers, not the R facilities (e.g. scan).

`asTree` this only applies when on passes a value for the `handlers` argument and is used then to determine whether the DOM tree should be returned or the `handlers` object.

`addAttributeNamespaces`
a logical value indicating whether to return the namespace in the names of the attributes within a node or to omit them. If this is `TRUE`, an attribute such as `xsi:type="xsd:string"` is reported with the name `xsi:type`. If it is `FALSE`, the name of the attribute is `type`.

`useInternalNodes`
a logical value indicating whether to call the converter functions with objects of class `XMLInternalNode` rather than `XMLNode`. This should make things faster as we do not convert the contents of the internal nodes to R explicit objects. Also, it allows one to access the parent and ancestor nodes. However, since the objects refer to volatile C-level objects, one cannot store these nodes for use in further computations within R. They "disappear" after the processing the XML document is completed.

If this argument is `TRUE` and no handlers are provided, the return value is a reference to the internal C-level document pointer. This can be used to do post-processing via XPath expressions using `getNodeSet`.

This is ignored when parsing an HTML document.

`isSchema` a logical value indicating whether the document is an XML schema (`TRUE`) and should be parsed as such using the built-in schema parser in libxml.

`fullNamespaceInfo`
a logical value indicating whether to provide the namespace URI and prefix on each node or just the prefix. The latter (`FALSE`) is currently the default as that was the original way the package behaved. However, using `TRUE` is more informative and we will make this the default in the future.

This is ignored when parsing an HTML document.

`encoding` a character string (scalar) giving the encoding for the document. This is optional as the document should contain its own encoding information. However, if it doesn't, the caller can specify this for the parser. If the XML/HTML document does specify its own encoding that value is used regardless of any value specified by the caller. (That's just the way it goes!) So this is to be used as a safety net in case the document does not have an encoding and the caller happens to know theactual encoding.

`useDotNames` a logical value indicating whether to use the newer format for identifying general element function handlers with the '.' prefix, e.g. .text, .comment, .startElement. If this is `FALSE`, then the older format text, comment, startElement, ... are used. This causes problems when there are indeed nodes named text or comment or startElement as a node-specific handler are confused with the corresponding general handler of the same name. Using `TRUE` means that your list of handlers should have names that use the '.' prefix for these general element handlers. This is the preferred way to write new code.

`xinclude` a logical value indicating whether to process nodes of the form `<xi:include xmlns:xi="https://www.w3.org/2001/XInclude">` to insert content from other parts of (potentially different) documents. `TRUE` means resolve the external references; `FALSE` means leave the node as is. Of course, one can process these

nodes oneself after document has been parse using handler functions or working on the DOM. Please note that the syntax for inclusion using XPointer is not the same as XPath and the results can be a little unexpected and confusing. See the libxml2 documentation for more details.

`addFinalizer` a logical value indicating whether the default finalizer routine should be registered to free the internal xmlDoc when R no longer has a reference to this external pointer object. This is only relevant when `useInternalNodes` is `TRUE`.

`error` a function that is invoked when the XML parser reports an error. When an error is encountered, this is called with 7 arguments. See `xmlStructuredStop` for information about these

If parsing completes and no document is generated, this function is called again with only argument which is a character vector of length 0. This gives the function an opportunity to report all the errors and raise an exception rather than doing this when it sees th first one.

This function can do what it likes with the information. It can raise an R error or let parser continue and potentially find further errors.

The default value of this argument supplies a function that cumulates the errors

If this is `NULL`, the default error handler function in the package `xmlStructuredStop` is invoked and this will raise an error in R at that time in R.

`isHTML` a logical value that allows this function to be used for parsing HTML documents. This causes validation and processing of a DTD to be turned off. This is currently experimental so that we can implement `htmlParse` with this same function.

`options` an integer value or vector of values that are combined (OR'ed) together to specify options for the XML parser. This is the same as the `options` parameter for `xmlParseDoc`.

`parentFirst` a logical value for use when we have handler functions and are traversing the tree. This controls whether we process the node before processing its children, or process the children before their parent node.

Details

The `handlers` argument is used similarly to those specified in xmlEventParse. When an XML tag (element) is processed, we look for a function in this collection with the same name as the tag's name. If this is not found, we look for one named `startElement`. If this is not found, we use the default built in converter. The same works for comments, entity references, cdata, processing instructions, etc. The default entries should be named `comment`, `startElement`, `externalEntity`, `processingInstruction`, `text`, `cdata` and `namespace`. All but the last should take the XMLnode as their first argument. In the future, other information may be passed via ..., for example, the depth in the tree, etc. Specifically, the second argument will be the parent node into which they are being added, but this is not currently implemented, so should have a default value (`NULL`).

The `namespace` function is called with a single argument which is an object of class `XMLNameSpace`. This contains

id the namespace identifier as used to qualify tag names;

uri the value of the namespace identifier, i.e. the URI identifying the namespace.

local a logical value indicating whether the definition is local to the document being parsed.

One should note that the `namespace` handler is called before the node in which the namespace definition occurs and its children are processed. This is different than the other handlers which are called after the child nodes have been processed.

Each of these functions can return arbitrary values that are then entered into the tree in place of the default node passed to the function as the first argument. This allows the caller to generate the nodes of the resulting document tree exactly as they wish. If the function returns `NULL`, the node is dropped from the resulting tree. This is a convenient way to discard nodes having processed their contents.

Value

By default (when `useInternalNodes` is `FALSE`, `getDTD` is `TRUE`, and no handler functions are provided), the return value is, an object of (S3) class `XMLDocument`. This has two fields named `doc` and `dtd` and are of class `DTDList` and `XMLDocumentContent` respectively.

If `getDTD` is `FALSE`, only the `doc` object is returned.

The `doc` object has three fields of its own: `file`, `version` and `children`.

`file` The (expanded) name of the file containing the XML.

`version` A string identifying the version of XML used by the document.

`children` A list of the XML nodes at the top of the document. Each of these is of class `XMLNode`. These are made up of 4 fields.

`name` The name of the element.

`attributes` For regular elements, a named list of XML attributes converted from the <tag x="1" y="abc">

`children` List of sub-nodes.

`value` Used only for text entries.

Some nodes specializations of `XMLNode`, such as `XMLComment`, `XMLProcessingInstruction`, `XMLEntityRef` are used.

If the value of the argument getDTD is TRUE and the document refers to a DTD via a top-level DOCTYPE element, the DTD and its information will be available in the `dtd` field. The second element is a list containing the external and internal DTDs. Each of these contains 2 lists - one for element definitions and another for entities. See `parseDTD`.

If a list of functions is given via `handlers`, this list is returned. Typically, these handler functions share state via a closure and the resulting updated data structures which contain the extracted and processed values from the XML document can be retrieved via a function in this handler list.

If `asTree` is `TRUE`, then the converted tree is returned. What form this takes depends on what the handler functions have done to process the XML tree.

If `useInternalNodes` is `TRUE` and no handlers are specified, an object of S3 class `XMLInternalDocument` is returned. This can be used in much the same ways as an `XMLDocument`, e.g. with `xmlRoot`, `docName` and so on to traverse the tree. It can also be used with XPath queries via `getNodeSet`, `xpathApply` and `doc["xpath-expression"]`.

If internal nodes are used and the internal tree returned directly, all the nodes are returned as-is and no attempt to trim white space, remove "empty" nodes (i.e. containing only white space), etc. is done. This is potentially quite expensive and so is not done generally, but should be done during the processing of the nodes. When using XPath queries, such nodes are easily identified and/or ignored and so do not cause any difficulties. They do become an issue when dealing with a node's chidren directly and so one can use simple filtering techniques such as `xmlChildren(node)[!xmlSApply(node, inherits, "XMLInternalTextNode")]` and even check the `xmlValue` to determine if it contains only white space. `xmlChildren(node)[!xmlSApply(node, function(x) inherit(x, "XMLInternalTextNode")] && trim(xmlValue(x)) == "")`

Note

Make sure that the necessary 3rd party libraries are available.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

http://xmlsoft.org, https://www.w3.org/XML/

See Also

xmlEventParse, `free` for releasing the memory when an `XMLInternalDocument` object is returned.

Examples

```
fileName <- system.file("exampleData", "test.xml", package="XML")
  # parse the document and return it in its standard format.

xmlTreeParse(fileName)

  # parse the document, discarding comments.

xmlTreeParse(fileName, handlers=list("comment"=function(x,...){NULL}), asTree = TRUE)

  # print the entities
invisible(xmlTreeParse(fileName,
           handlers=list(entity=function(x) {
                                cat("In entity",x$name, x$value,"\n")
                                x}
                               ), asTree = TRUE
                         )
        )

# Parse some XML text.
# Read the text from the file
xmlText <- paste(readLines(fileName), "\n", collapse="")
```

```
print(xmlText)
xmlTreeParse(xmlText, asText=TRUE)


   # with version 1.4.2 we can pass the contents of an XML
   # stream without pasting them.
xmlTreeParse(readLines(fileName), asText=TRUE)


# Read a MathML document and convert each node
# so that the primary class is
#   <name of tag>MathML
# so that we can use method  dispatching when processing
# it rather than conditional statements on the tag name.
# See plotMathML() in examples/.
fileName <- system.file("exampleData", "mathml.xml",package="XML")
m <- xmlTreeParse(fileName,
                  handlers=list(
                   startElement = function(node){
                   cname <- paste(xmlName(node),"MathML", sep="",collapse="")
                   class(node) <- c(cname, class(node));
                   node
                 }))



 # In this example, we extract _just_ the names of the
 # variables in the mtcars.xml file.
 # The names are the contents of the <variable>
 # tags. We discard all other tags by returning NULL
 # from the startElement handler.
 #
 # We cumulate the names of variables in a character
 # vector named 'vars'.
 # We define this within a closure and define the
 # variable function within that closure so that it
 # will be invoked when the parser encounters a <variable>
 # tag.
 # This is called with 2 arguments: the XMLNode object (containing
 # its children) and the list of attributes.
 # We get the variable name via call to xmlValue().

 # Note that we define the closure function in the call and then
 # create an instance of it by calling it directly as
 #   (function() {...})()

 # Note that we can get the names by parsing
 # in the usual manner and the entire document and then executing
 # xmlSApply(xmlRoot(doc)[[1]], function(x) xmlValue(x[[1]]))
 # which is simpler but is more costly in terms of memory.
fileName <- system.file("exampleData", "mtcars.xml", package="XML")
doc <- xmlTreeParse(fileName,  handlers = (function() {
```

```
                                  vars <- character(0) ;
                                 list(variable=function(x, attrs) {
                                                vars <<- c(vars, xmlValue(x[[1]]));
                                                NULL},
                                      startElement=function(x,attr){
                                                   NULL
                                                 },
                                      names = function() {
                                                 vars
                                              }
                                    )
                                 })()
                     )

  # Here we just print the variable names to the console
  # with a special handler.
 doc <- xmlTreeParse(fileName, handlers = list(
                                  variable=function(x, attrs) {
                                            print(xmlValue(x[[1]])); TRUE
                                           }), asTree=TRUE)


  # This should raise an error.
  try(xmlTreeParse(
            system.file("exampleData", "TestInvalid.xml", package="XML"),
            validate=TRUE))

## Not run:
 # Parse an XML document directly from a URL.
 # Requires Internet access.
 xmlTreeParse("https://www.omegahat.net/Scripts/Data/mtcars.xml", asText=TRUE)

## End(Not run)

  counter = function() {
              counts = integer(0)
              list(startElement = function(node) {
                                    name = xmlName(node)
                                    if(name %in% names(counts))
                                         counts[name] <<- counts[name] + 1
                                    else
                                         counts[name] <<- 1
                                  },
                   counts = function() counts)
            }

   h = counter()
   xmlParse(system.file("exampleData", "mtcars.xml", package="XML"),  handlers = h)
   h$counts()



 f = system.file("examples", "index.html", package = "XML")
```

```
htmlTreeParse(readLines(f), asText = TRUE)
htmlTreeParse(readLines(f))

 # Same as
htmlTreeParse(paste(readLines(f), collapse = "\n"), asText = TRUE)


getLinks = function() {
      links = character()
      list(a = function(node, ...) {
                  links <<- c(links, xmlGetAttr(node, "href"))
                  node
              },
           links = function()links)
    }

h1 = getLinks()
htmlTreeParse(system.file("examples", "index.html", package = "XML"),
              handlers = h1)
h1$links()

h2 = getLinks()
htmlTreeParse(system.file("examples", "index.html", package = "XML"),
             handlers = h2, useInternalNodes = TRUE)
all(h1$links() == h2$links())

 # Using flat trees
tt = xmlHashTree()
f = system.file("exampleData", "mtcars.xml", package="XML")
xmlTreeParse(f, handlers = list(.startElement = tt[[".addNode"]]))
xmlRoot(tt)



doc = xmlTreeParse(f, useInternalNodes = TRUE)

sapply(getNodeSet(doc, "//variable"), xmlValue)

#free(doc)


 # character set encoding for HTML
f = system.file("exampleData", "9003.html", package = "XML")
  # we specify the encoding
d = htmlTreeParse(f, encoding = "UTF-8")
  # get a different result if we do not specify any encoding
d.no = htmlTreeParse(f)
  # document with its encoding in the HEAD of the document.
d.self = htmlTreeParse(system.file("exampleData", "9003-en.html",package = "XML"))
  # XXX want to do a test here to see the similarities between d and
  # d.self and differences between d.no
```

```
 # include
f = system.file("exampleData", "nodes1.xml", package = "XML")
xmlRoot(xmlTreeParse(f, xinclude = FALSE))
xmlRoot(xmlTreeParse(f, xinclude = TRUE))

f = system.file("exampleData", "nodes2.xml", package = "XML")
xmlRoot(xmlTreeParse(f, xinclude = TRUE))

 # Errors
 try(xmlTreeParse("<doc><a> & < <?pi > </doc>"))

   # catch the error by type.
tryCatch(xmlTreeParse("<doc><a> & < <?pi > </doc>"),
               "XMLParserErrorList" = function(e) {
                    cat("Errors in XML document\n", e$message, "\n")
                                                 })

   #  terminate on first error
 try(xmlTreeParse("<doc><a> & < <?pi > </doc>", error = NULL))

   #  see xmlErrorCumulator in the XML package


f = system.file("exampleData", "book.xml", package = "XML")
doc.trim = xmlInternalTreeParse(f, trim = TRUE)
doc = xmlInternalTreeParse(f, trim = FALSE)
xmlSApply(xmlRoot(doc.trim), class)
    # note the additional XMLInternalTextNode objects
xmlSApply(xmlRoot(doc), class)


top = xmlRoot(doc)
textNodes = xmlSApply(top, inherits, "XMLInternalTextNode")
sapply(xmlChildren(top)[textNodes], xmlValue)


   # Storing nodes
 f = system.file("exampleData", "book.xml", package = "XML")
 titles = list()
 xmlTreeParse(f, handlers = list(title = function(x)
                                titles[[length(titles) + 1]] <<- x))
 sapply(titles, xmlValue)
 rm(titles)
```

`xmlValue` *Extract or set the contents of a leaf XML node*

Description

Some types of XML nodes have no children nodes, but are leaf nodes and simply contain text. Examples are `XMLTextMode`, `XMLProcessingInstruction`. This function provides access to their

raw contents. This has been extended to operate recursivel on arbitrary XML nodes that contain a single text node.

Usage

```
xmlValue(x, ignoreComments = FALSE, recursive = TRUE,
          encoding = getEncoding(x), trim = FALSE)
```

Arguments

x	the XMLNode object whose contents are to be returned.
ignoreComments	a logical value which, if TRUE does not include the text in XML comment nodes. If this is FALSE, the text in the comments is part of the return value.
recursive	a logical value indicating whether to process all sub-nodes (TRUE) or only the text nodes within the node x.
encoding	experimental functionality and parameter related to encoding.
trim	a logical value controlling whether we remove leading or trailing white space when returning the string value

Value

The object stored in the value slot of the XMLNode object. This is typically a string.

Author(s)

Duncan Temple Lang

References

https://www.w3.org/XML/, http://www.jclark.com/xml/

See Also

xmlChildren xmlName xmlAttrs xmlNamespace

Examples

```
node <- xmlNode("foo", "Some text")
xmlValue(node)

xmlValue(xmlTextNode("some more raw text"))

 # Setting the xmlValue().
a = newXMLNode("a")
xmlValue(a) = "the text"
xmlValue(a) = "different text"

a = newXMLNode("x", "bob")
xmlValue(a) = "joe"
```

```
b = xmlNode("bob")
xmlValue(b) = "Foo"
xmlValue(b) = "again"

b = newXMLNode("bob", "some text")
xmlValue(b[[1]]) = "change"
b
```

Index

∗ **Annotated code**
  xmlSource, 130
∗ **Error handling**
  xmlStopParser, 133
∗ **IO**
  [.XMLNode, 4
  [<-.XMLNode, 5
  addChildren, 6
  addNode, 10
  append.xmlNode, 11
  asXMLTreeNode, 14
  catalogLoad, 15
  catalogResolve, 17
  coerceNodes, 18
  compareXMLDocs, 19
  docName, 20
  Doctype, 21
  ensureNamespace, 23
  findXInclude, 24
  free, 25
  getEncoding, 28
  getHTMLLinks, 29
  getLineNumber, 31
  getNodeSet, 32
  getRelativeURL, 40
  getSibling, 41
  getXIncludes, 43
  getXMLErrors, 44
  isXMLString, 45
  libxmlVersion, 47
  newXMLDoc, 51
  newXMLNamespace, 57
  parseURI, 58
  parseXMLAndAdd, 59
  print.XMLAttributeDef, 61
  processXInclude, 63
  readHTMLList, 64
  readHTMLTable, 65
  readKeyValueDB, 68
  readSolrDoc, 69
  removeXMLNamespaces, 70
  replaceNodeWithChildren, 71
  saveXML, 72
  toHTML, 80
  xmlAttrs, 85
  xmlCleanNamespaces, 87
  xmlClone, 88
  xmlElementsByTagName, 92
  xmlElementSummary, 94
  xmlEventHandler, 95
  xmlEventParse, 96
  xmlHandler, 105
  xmlHashTree, 106
  xmlNamespaceDefinitions, 112
  xmlOutputBuffer, 117
  xmlParent, 119
  xmlParserContextFunction, 123
  xmlSchemaValidate, 125
  xmlSerializeHook, 128
  xmlSource, 130
  xmlStopParser, 133
  xmlStructuredStop, 135
  xmlToDataFrame, 136
  xmlToList, 138
  xmlToS4, 139
  xmlTree, 140
  xmlTreeParse, 143
∗ **Literate Programming**
  xmlSource, 130
∗ **Mixed language**
  xmlSource, 130
∗ **Solr**
  readSolrDoc, 69
∗ **URI**
  parseURI, 58
∗ **Web**
  parseURI, 58
∗ **XML**

addChildren, 6
asXMLTreeNode, 14
catalogResolve, 17
coerceNodes, 18
docName, 20
ensureNamespace, 23
getLineNumber, 31
getRelativeURL, 40
getXIncludes, 43
isXMLString, 45
newXMLNamespace, 57
processXInclude, 63
readKeyValueDB, 68
toHTML, 80
xmlClone, 88
xmlElementSummary, 94
xmlHashTree, 106
xmlNamespaceDefinitions, 112
xmlParseDoc, 121
xmlParserContextFunction, 123
xmlSchemaValidate, 125

∗ **classes**
Doctype-class, 22
SAXState-class, 74
schema-class, 76
XMLAttributes-class, 83
XMLCodeFile-class, 89
XMLInternalDocument-class, 109
XMLNode-class, 116

∗ **data exchange**
toHTML, 80

∗ **data**
readHTMLTable, 65
xmlParseDoc, 121
xmlSearchNs, 127
xmlToList, 138

∗ **document tree**
addChildren, 6

∗ **error handling**
xmlStructuredStop, 135

∗ **external memory**
free, 25

∗ **file**
[.XMLNode, 4
[<-.XMLNode, 5
append.xmlNode, 11
asXMLNode, 13
genericSAXHandlers, 26
getNodeSet, 32
length.XMLNode, 46
names.XMLNode, 50
print.XMLAttributeDef, 61
saveXML, 72
startElement.SAX, 78
supportsExpat, 79
toString.XMLNode, 81
xmlApply, 82
xmlAttributeType, 84
xmlAttrs, 85
xmlChildren, 86
xmlDOMApply, 91
xmlElementsByTagName, 92
xmlEventHandler, 95
xmlEventParse, 96
xmlGetAttr, 104
xmlHandler, 105
xmlName, 110
xmlNamespace, 111
xmlNode, 114
xmlOutputBuffer, 117
xmlParent, 119
xmlRoot, 124
xmlSize, 129
xmlTreeParse, 143
xmlValue, 153

∗ **meta-computing**
xmlToS4, 139

∗ **meta-programming**
makeClassTemplate, 48

∗ **programming**
addChildren, 6
coerceNodes, 18
docName, 20
getChildrenStrings, 27
getHTMLLinks, 29
getRelativeURL, 40
getXMLErrors, 44
makeClassTemplate, 48
newXMLNamespace, 57
processXInclude, 63
readHTMLList, 64
setXMLNamespace, 77
toHTML, 80
xmlCleanNamespaces, 87
xmlClone, 88
xmlParserContextFunction, 123

xmlSearchNs, 127
xmlSource, 130
xmlStopParser, 133
xmlStructuredStop, 135
xmlToS4, 139
∗ **reflection**
makeClassTemplate, 48
∗ **schema**
xmlSchemaValidate, 125
∗ **serialization**
toHTML, 80
∗ **streaming data**
xmlStopParser, 133
∗ **tree**
addNode, 10
∗ **validation**
xmlSchemaValidate, 125
.InitSAXMethods, *27*
.InitSAXMethods *(*startElement.SAX*)*, 78
[,XMLAttributes-method *(*XMLAttributes-class*)*, 83
[.XMLNode, 4, *6*, *12*, *83*
[<-.XMLNode, 5
[[,XMLCodeFile,ANY-method *(*XMLCodeFile-class*)*, 89
[[,XMLCodeFile-method *(*XMLCodeFile-class*)*, 89
[[.XMLDocumentContent *(*[.XMLNode*)*, 4
[[.XMLInternalElementNode *(*[.XMLNode*)*, 4
[[.XMLNode, *6*, *12*, *125*
[[.XMLNode *(*[.XMLNode*)*, 4
[[<-.XMLNode *(*[<-.XMLNode*)*, 5
$,libxmlTypeTable-method *(*schema-class*)*, 76
$,xmlSchemaRef-method *(*schema-class*)*, 76
$<-,libxmlTypeTable-method *(*schema-class*)*, 76

addAttributes *(*addChildren*)*, 6
addAttributes,XMLInternalElementNode-method *(*addChildren*)*, 6
addAttributes,XMLNode-method *(*addChildren*)*, 6
addChildren, 6, *42*, *53*, *60*, *85*, *115*
addChildren,XMLInternalNode-method *(*addChildren*)*, 6
addChildren,XMLNode-method *(*addChildren*)*, 6
addNode, *6*, 10
addSibling *(*getSibling*)*, 41
append.XMLNode *(*append.xmlNode*)*, 11
append.xmlNode, *6*, 11, *118*
apply, *82*
as.data.frame, *66*
asXMLNode, 13, *115*
asXMLTreeNode, *11*, 14

catalogAdd *(*catalogLoad*)*, 15
catalogClearTable *(*catalogLoad*)*, 15
catalogDump *(*catalogLoad*)*, 15
catalogLoad, 15
catalogResolve, *15*, *16*, 17
character, *83*, *90*
coerce, *22*
coerce,character,Currency-method *(*readHTMLTable*)*, 65
coerce,character,FormattedInteger-method *(*readHTMLTable*)*, 65
coerce,character,FormattedNumber-method *(*readHTMLTable*)*, 65
coerce,character,Percent-method *(*readHTMLTable*)*, 65
coerce,character,XMLCodeDoc-method *(*XMLCodeFile-class*)*, 89
coerce,character,XMLCodeFile-method *(*XMLCodeFile-class*)*, 89
coerce,character,XMLNamespaceDefinitions-method *(*xmlNamespaceDefinitions*)*, 112
coerce,Doctype,character-method *(*Doctype*)*, 21
coerce,libxmlTypeTable,list-method *(*schema-class*)*, 76
coerce,NULL,XMLNamespaceDefinitions-method *(*xmlNamespaceDefinitions*)*, 112
coerce,URI,character-method *(*parseURI*)*, 58
coerce,vector,XMLInternalNode-method *(*newXMLDoc*)*, 51
coerce,XMLAbstractDocument,XMLAbstractNode-method *(*coerceNodes*)*, 18
coerce,XMLAbstractNode,character-method *(*XMLNode-class*)*, 116
coerce,XMLAbstractNode,Date-method *(*XMLNode-class*)*, 116
coerce,XMLAbstractNode,integer-method *(*XMLNode-class*)*, 116
coerce,XMLAbstractNode,logical-method *(*XMLNode-class*)*, 116

coerce,XMLAbstractNode,numeric-method (XMLNode-class), 116
coerce,XMLAbstractNode,POSIXct-method (XMLNode-class), 116
coerce,XMLAbstractNode,URL-method (XMLNode-class), 116
coerce,XMLCodeFile,XMLCodeDoc-method (XMLCodeFile-class), 89
coerce,XMLDocument,XMLInternalDocument-method (XMLInternalDocument-class), 109
coerce,XMLHashTreeNode,XMLHashTree-method (coerceNodes), 18
coerce,XMLInternalDocument,character-method (saveXML), 72
coerce,XMLInternalDocument,XMLHashTree-method (coerceNodes), 18
coerce,XMLInternalDocument,XMLInternalNode-method (XMLInternalDocument-class), 109
coerce,XMLInternalDOM,character-method (saveXML), 72
coerce,XMLInternalNode,character-method (saveXML), 72
coerce,XMLInternalNode,XMLHashTree-method (coerceNodes), 18
coerce,XMLInternalNode,XMLInternalDocument-method (XMLInternalDocument-class), 109
coerce,XMLInternalNode,XMLNode-method (asXMLNode), 13
coerce,XMLInternalTextNode,character-method (xmlValue), 153
coerce,XMLNamespace,character-method (xmlNamespaceDefinitions), 112
coerce,XMLNamespaceDefinition,character-method (xmlNamespaceDefinitions), 112
coerce,XMLNamespaceDefinitions,character-method (xmlNamespaceDefinitions), 112
coerce,XMLNamespaceRef,character-method (xmlSearchNs), 127
coerce,XMLNode,XMLInternalNode-method (coerceNodes), 18
coerceNodes, 18
comment.SAX, *27*
comment.SAX (startElement.SAX), 78
comment.SAX,ANY,SAXState-method (startElement.SAX), 78
COMPACT (xmlParseDoc), 121
compareXMLDocs, 19

data.frameRowLabels, *83*
docName, 20, *148*
docName,NULL-method (docName), 20
docName,XMLDocument-method (docName), 20
docName,XMLDocumentContent-method (docName), 20
docName,XMLHashTree-method (docName), 20
docName,XMLHashTreeNode-method (docName), 20
docName,XMLInternalDocument-method (docName), 20
docName,XMLInternalNode-method (docName), 20
docName,XMLNode-method (docName), 20
docName<- (docName), 20
docName<-,XMLHashTree-method (docName), 20
docName<-,XMLInternalDocument-method (docName), 20
Doctype, 21, *22*, *23*
Doctype-class, 22
DTDATTR (xmlParseDoc), 121
DTDLOAD (xmlParseDoc), 121
DTDVALID (xmlParseDoc), 121

Encoding, *28*
endElement.SAX, *27*
endElement.SAX (startElement.SAX), 78
endElement.SAX,ANY,SAXState-method (startElement.SAX), 78
ensureNamespace, 23
entityDeclaration.SAX, *27*
entityDeclaration.SAX (startElement.SAX), 78
entityDeclaration.SAX,ANY,ANY,ANY,ANY,ANY,SAXState-method (startElement.SAX), 78
ExternalReference-class (schema-class), 76

findXInclude, 24, *31*
FormattedInteger-class (readHTMLTable), 65
FormattedNumber-class (readHTMLTable), 65
free, 25, *149*

free,XMLInternalDocument-method *(free)*, 25

genericSAXHandlers, 26
getChildrenStrings, 27
getDefaultNamespace *(xmlNamespaceDefinitions)*, 112
getEncoding, 28
getEncoding,ANY-method *(getEncoding)*, 28
getEncoding,XMLInternalDocument-method *(getEncoding)*, 28
getEncoding,XMLInternalNode-method *(getEncoding)*, 28
getHTMLExternalFiles, *44*
getHTMLExternalFiles *(getHTMLLinks)*, 29
getHTMLLinks, 29
getLineNumber, 31
getNativeSymbolInfo, *34*, *52*, *89*, *120*
getNodeLocation *(getLineNumber)*, 31
getNodePosition *(getLineNumber)*, 31
getNodeSet, *19*, *31*, 32, *66*, *70*, *109*, *112, 113*, *137*, *139*, *143*, *146*, *148*
getRelativeURL, 40, *59*
getSibling, 41
getXIncludes, *30*, 43
getXMLErrors, 44
grep, *145*

HTMLInternalDocument-class *(XMLInternalDocument-class)*, 109
htmlParse, *64–66*
htmlParse *(xmlTreeParse)*, 143
htmlTreeParse, *44*, *109*, *135, 136*
htmlTreeParse *(xmlTreeParse)*, 143
HUGE *(xmlParseDoc)*, 121

isXMLString, 45

lapply, *11*, *32*, *34*, *82, 83*
length, *129*
length.XMLNode, 46
libxmlFeatures *(libxmlVersion)*, 47
libxmlTypeTable-class *(schema-class)*, 76
libxmlVersion, 47
list, *105*

makeClassTemplate, 48, *139, 140*
matchNamespaces *(getNodeSet)*, 32

names, *99*
names,libxmlTypeTable-method *(schema-class)*, 76
names,xmlSchemaRef-method *(schema-class)*, 76
names.XMLNode, 50
newHTMLDoc *(newXMLDoc)*, 51
newXMLCDataNode, *115*
newXMLCDataNode *(newXMLDoc)*, 51
newXMLCommentNode, *115*
newXMLCommentNode *(newXMLDoc)*, 51
newXMLDoc, *21*, 51, *73*, *89*, *141*
newXMLDTDNode, *140*
newXMLDTDNode *(newXMLDoc)*, 51
newXMLNamespace, *24*, 57, *58*, *70*, *77*
newXMLNode, *8*, *19*, *24*, *60*, *63*, *73*, *83*, *89*, *115, 116*, *120*, *127*, *141*
newXMLNode *(newXMLDoc)*, 51
newXMLPINode, *115*
newXMLPINode *(newXMLDoc)*, 51
newXMLTextNode, *7*
newXMLTextNode *(newXMLDoc)*, 51
NOBASEFIX *(xmlParseDoc)*, 121
NOBLANKS *(xmlParseDoc)*, 121
NOCDATA *(xmlParseDoc)*, 121
NODICT *(xmlParseDoc)*, 121
NOENT *(xmlParseDoc)*, 121
NOERROR *(xmlParseDoc)*, 121
NONET *(xmlParseDoc)*, 121
NOWARNING *(xmlParseDoc)*, 121
NOXINCNODE *(xmlParseDoc)*, 121
NSCLEAN *(xmlParseDoc)*, 121

OLD10 *(xmlParseDoc)*, 121
oldClass, *109*
OLDSAX *(xmlParseDoc)*, 121

parseDTD, *84*, *117*, *148*
parseURI, *41*, 58
parseXMLAndAdd, 59
PEDANTIC *(xmlParseDoc)*, 121
Percent-class *(readHTMLTable)*, 65
print, *61*
print.XMLAttributeDef, 61
print.XMLCDataNode *(print.XMLAttributeDef)*, 61
print.XMLComment *(print.XMLAttributeDef)*, 61

print.XMLElementContent
(print.XMLAttributeDef), 61
print.XMLElementDef
(print.XMLAttributeDef), 61
print.XMLEntity
(print.XMLAttributeDef), 61
print.XMLEntityRef
(print.XMLAttributeDef), 61
print.XMLNode (print.XMLAttributeDef),
61
print.XMLOrContent
(print.XMLAttributeDef), 61
print.XMLProcessingInstruction
(print.XMLAttributeDef), 61
print.XMLSequenceContent
(print.XMLAttributeDef), 61
print.XMLTextNode
(print.XMLAttributeDef), 61
processingInstruction.SAX, *27*
processingInstruction.SAX
(startElement.SAX), 78
processingInstruction.SAX,ANY,ANY,SAXState-method
(startElement.SAX), 78
processXInclude, 63

read.table, *137*
readHTMLList, 64
readHTMLList,character-method
(readHTMLList), 64
readHTMLList,HTMLInternalDocument-method
(readHTMLList), 64
readHTMLList,XMLInternalNode-method
(readHTMLList), 64
readHTMLTable, *64*, 65, *65*
readHTMLTable,character-method
(readHTMLTable), 65
readHTMLTable,HTMLInternalDocument-method
(readHTMLTable), 65
readHTMLTable,XMLInternalElementNode-method
(readHTMLTable), 65
readKeyValueDB, 68, *70*
readKeyValueDB,AsIs-method
(readKeyValueDB), 68
readKeyValueDB,character-method
(readKeyValueDB), 68
readKeyValueDB,XMLInternalDocument-method
(readKeyValueDB), 68
readKeyValueDB,XMLInternalNode-method
(readKeyValueDB), 68
readLines, *97*
readRDS, *128*
readSolrDoc, 69, *69*
readSolrDoc,AsIs-method (readSolrDoc),
69
readSolrDoc,character-method
(readSolrDoc), 69
readSolrDoc,XMLInternalDocument-method
(readSolrDoc), 69
readSolrDoc,XMLInternalNode-method
(readSolrDoc), 69
RECOVER (xmlParseDoc), 121
removeAttributes (addChildren), 6
removeAttributes,XMLInternalElementNode-method
(addChildren), 6
removeAttributes,XMLNode-method
(addChildren), 6
removeChildren (addChildren), 6
removeNodes, *42*
removeNodes (addChildren), 6
removeXMLNamespaces, 70, *77*
removeXMLNamespaces,XMLInternalDocument-method
(removeXMLNamespaces), 70
removeXMLNamespaces,XMLInternalElementNode-method
(removeXMLNamespaces), 70
removeXMLNamespaces,XMLInternalNode-method
(removeXMLNamespaces), 70
replaceNodes, *42*
replaceNodes (addChildren), 6
replaceNodeWithChildren, 71
RXMLNode-class (XMLNode-class), 116

sapply, *32*, *34*, *82, 83*, *133*
saveRDS, *128*
saveXML, *22, 23*, *54*, 72, *141*
saveXML,HTMLInternalDocument-method
(saveXML), 72
saveXML,XMLFlatTree-method (saveXML), 72
saveXML,XMLInternalDocument-method
(saveXML), 72
saveXML,XMLInternalDOM-method
(saveXML), 72
saveXML,XMLInternalNode-method
(saveXML), 72
saveXML,XMLNode-method (saveXML), 72
saveXML,XMLOutputStream-method
(saveXML), 72
saveXML.XMLInternalDocument (saveXML),
72

saveXML.XMLInternalDOM (saveXML), 72
saveXML.XMLInternalNode (saveXML), 72
saveXML.XMLNode (saveXML), 72
saveXML.XMLOutputStream (saveXML), 72
SAX1 (xmlParseDoc), 121
SAXState-class, 74
scan, *145*
schema-class, 76
SchemaAttributeGroupTable-class (schema-class), 76
SchemaAttributeTable-class (schema-class), 76
SchemaElementTable-class (schema-class), 76
SchemaNotationTable-class (schema-class), 76
SchemaTypeTable-class (schema-class), 76
schemaValidationErrorHandler (xmlSchemaValidate), 125
setXMLNamespace, 77
show,XMLAttributes-method (XMLAttributes-class), 83
show,XMLSchemaValidationResults-method (schema-class), 76
simpleError, *135*
source, *90*, *130*, *132*
source,XMLCodeFile-method (XMLCodeFile-class), 89
startElement.SAX, *27*, 78
startElement.SAX,ANY,ANY,SAXState-method (startElement.SAX), 78
stop, *136*
SuperClassMethod, *83*
supportsExpat, 79, *98*
supportsLibxml (supportsExpat), 79

text.SAX (startElement.SAX), 78
text.SAX,ANY,SAXState-method (startElement.SAX), 78
toHTML, 80
toHTML,call-method (toHTML), 80
toHTML,matrix-method (toHTML), 80
toHTML,vector-method (toHTML), 80
toString.XMLNode, 81

URI-class (parseURI), 58

vector, *83*, *90*

XINCLUDE (xmlParseDoc), 121
xml (isXMLString), 45
XMLAbstractDocument-class (XMLInternalDocument-class), 109
XMLAbstractNode-class (XMLNode-class), 116
xmlAncestors (xmlParent), 119
xmlApply, *50*, 82, *139*
XMLAttributeDeclNode-class (XMLNode-class), 116
XMLAttributes-class, 83
xmlAttributeType, 84
xmlAttrs, *5*, *50*, *83*, 85, *104*, *105*, *110*, *112*, *113*, *115*, *129*, *154*
xmlAttrs<- (xmlAttrs), 85
xmlAttrs<-,XMLInternalElementNode-method (xmlAttrs), 85
xmlAttrs<-,XMLInternalNode (xmlAttrs), 85
xmlAttrs<-,XMLNode (xmlAttrs), 85
xmlAttrs<-,XMLNode-method (xmlAttrs), 85
xmlCDataNode (xmlNode), 114
xmlChildren, *42*, *47*, *83*, *85*, 86, *87*, *93*, *110*, *112*, *115*, *121*, *129*, *139*, *154*
xmlChildren<- (xmlChildren), 86
xmlChildren<-,ANY-method (xmlChildren), 86
xmlChildren<-,XMLInternalNode-method (xmlChildren), 86
xmlCleanNamespaces, 87
xmlClone, 88
xmlClone,XMLInternalDocument-method (xmlClone), 88
xmlClone,XMLInternalNode-method (xmlClone), 88
XMLCodeDoc-class (XMLCodeFile-class), 89
xmlCodeFile (XMLCodeFile-class), 89
XMLCodeFile-class, 89
xmlCommentNode (xmlNode), 114
xmlDeserializeHook (xmlSerializeHook), 128
xmlDoc (newXMLDoc), 51
XMLDocumentFragNode-class (XMLNode-class), 116
XMLDocumentNode-class (XMLNode-class), 116
XMLDocumentTypeNode-class (XMLNode-class), 116

xmlDOMApply, 91
XMLDTDNode-class (XMLNode-class), 116
xmlElementsByTagName, 92
xmlElementSummary, 94
XMLEntityDeclNode-class (XMLNode-class), 116
xmlErrorCumulator (xmlStructuredStop), 135
xmlEventHandler, 95
xmlEventParse, *26, 27*, *63*, *75*, *79*, *94, 95*, 96, *96*, *105, 106*, *123*, *133, 134*, *147*, *149*
xmlGetAttr, 104, *113*
xmlHandler, 105
xmlHashTree, *11*, *14*, 106, *118*, *120*
XMLInternalCDataNode-class (XMLNode-class), 116
XMLInternalCommentNode-class (XMLNode-class), 116
XMLInternalDocument-class, 109
XMLInternalElementNode-class (XMLNode-class), 116
XMLInternalNode-class (XMLNode-class), 116
XMLInternalPINode-class (XMLNode-class), 116
XMLInternalTextNode-class (XMLNode-class), 116
xmlInternalTreeParse, *21*, *41*, *63*, *123*, *126*, *136*
xmlInternalTreeParse (xmlTreeParse), 143
xmlName, *50*, *85*, 110, *112*, *115*, *129*, *154*
xmlName<- (xmlName), 110
xmlNamespace, 111, *115*, *154*
XMLNamespace-class (xmlNamespace), 111
xmlNamespace.character (xmlNamespace), 111
xmlNamespace.XMLInternalNode (xmlNamespace), 111
xmlNamespace.XMLNode (xmlNamespace), 111
xmlNamespace<- (xmlNamespace), 111
xmlNamespace<-,XMLInternalNode-method (xmlNamespace), 111
XMLNamespaceDeclNode-class (XMLNode-class), 116
xmlNamespaceDefinitions, *57*, 112, *112*
XMLNamespaceDefinitions-class (XMLNode-class), 116
xmlNamespaces (xmlNamespaceDefinitions), 112
xmlNamespaces<- (xmlNamespaceDefinitions), 112
xmlNamespaces<-,XMLInternalNode-method (xmlNamespaceDefinitions), 112
xmlNamespaces<-,XMLNode-method (xmlNamespaceDefinitions), 112
xmlNativeTreeParse (xmlTreeParse), 143
xmlNode, *13*, *81*, 114, *116*, *118*, *121*
XMLNode-class, 116
xmlOutputBuffer, *73*, *107*, 117, *140, 141*
xmlOutputDOM, *73*, *107*, *140, 141*
xmlOutputDOM (xmlOutputBuffer), 117
xmlParent, *109*, 119
xmlParent,XMLHashTreeNode-method (xmlParent), 119
xmlParent,XMLInternalNode-method (xmlParent), 119
xmlParent,XMLTreeNode-method (xmlParent), 119
xmlParent.XMLInternalNode (xmlParent), 119
xmlParent<- (addChildren), 6
xmlParse, *18, 19*, *25*, *31*, *41*, *46*, *60*, *69, 70*, *83*, *87–89*, *113*, *121–123*, *137*
xmlParse (xmlTreeParse), 143
xmlParseDoc, 121, *147*
xmlParserContextFunction, 123
xmlParseString (isXMLString), 45
xmlPINode (xmlNode), 114
xmlRoot, *83*, *91*, 124, *139*, *148*
xmlSApply, *50*
xmlSApply (xmlApply), 82
xmlSchemaAttributeGroupRef-class (schema-class), 76
xmlSchemaAttributeRef-class (schema-class), 76
xmlSchemaElementRef-class (schema-class), 76
xmlSchemaNotationRef-class (schema-class), 76
xmlSchemaParse, *126*
xmlSchemaParse (xmlTreeParse), 143
xmlSchemaRef-class (schema-class), 76
xmlSchemaTypeRef-class (schema-class), 76
xmlSchemaValidate, *76*, 125
xmlSearchNs, 127

xmlSerializeHook, 128
xmlSize, *6*, *46*, *47*, *85*, *87*, 129
xmlSource, *90*, 130
xmlSource,character-method *(*xmlSource*)*, 130
xmlSource,XMLInternalDocument-method *(*xmlSource*)*, 130
xmlSource,XMLNodeSet-method *(*xmlSource*)*, 130
xmlSourceFunctions *(*xmlSource*)*, 130
xmlSourceFunctions,character-method *(*xmlSource*)*, 130
xmlSourceFunctions,XMLInternalDocument-method *(*xmlSource*)*, 130
xmlSourceSection *(*xmlSource*)*, 130
xmlSourceSection,character-method *(*xmlSource*)*, 130
xmlSourceSection,XMLInternalDocument-method *(*xmlSource*)*, 130
xmlSourceThread *(*xmlSource*)*, 130
xmlSourceThread,character-method *(*xmlSource*)*, 130
xmlSourceThread,list-method *(*xmlSource*)*, 130
xmlSourceThread,XMLInternalDocument-method *(*xmlSource*)*, 130
xmlStopParser, *101*, 133
XMLString-class *(*isXMLString*)*, 45
xmlStructuredStop, 135, *147*
xmlTextNode, *7*, *13*, *118*
xmlTextNode *(*xmlNode*)*, 114
xmlToDataFrame, *69*, *70*, 136
xmlToDataFrame,ANY,ANY,ANY,ANY,list-method *(*xmlToDataFrame*)*, 136
xmlToDataFrame,ANY,ANY,ANY,ANY,XMLInternalNodeList-method *(*xmlToDataFrame*)*, 136
xmlToDataFrame,ANY,ANY,ANY,ANY,XMLNodeSet-method *(*xmlToDataFrame*)*, 136
xmlToDataFrame,character,ANY,ANY,ANY,ANY-method *(*xmlToDataFrame*)*, 136
xmlToDataFrame,list,ANY,ANY,ANY,ANY-method *(*xmlToDataFrame*)*, 136
xmlToDataFrame,XMLInternalDocument,ANY,ANY,ANY,missing-method *(*xmlToDataFrame*)*, 136
xmlToDataFrame,XMLInternalElementNode,ANY,ANY,ANY,ANY-method *(*xmlToDataFrame*)*, 136
xmlToDataFrame,XMLInternalNodeList,ANY,ANY,ANY,ANY-method *(*xmlToDataFrame*)*, 136
xmlToDataFrame,XMLNodeSet,ANY,ANY,ANY,ANY-method *(*xmlToDataFrame*)*, 136
xmlToList, *69*, *70*, 138
xmlToS4, *49*, 139
xmlToS4,XMLInternalNode-method *(*xmlToS4*)*, 139
xmlTree, *8*, *51*, *55*, *72*, *107*, *116*, *118*, *120*, 140
XMLTreeNode-class *(*XMLNode-class*)*, 116
xmlTreeParse, *18*, *21*, *26*, *35*, *41*, *44–46*, *62*, *63*, *81*, *87*, *91–93*, *96*, *99–101*, *106*, *107*, *109*, *110*, *113*, *115–117*, *120*, *121*, *125*, *129*, *131*, *133*, *135*, *136*, *139*, 143
xmlValue, *28*, *112*, *149*, 153
xmlValue<- *(*xmlValue*)*, 153
xmlValue<-,XMLAbstractNode-method *(*xmlValue*)*, 153
xmlValue<-,XMLInternalTextNode-method *(*xmlValue*)*, 153
xmlValue<-,XMLTextNode-method *(*xmlValue*)*, 153
XMLXIncludeEndNode-class *(*XMLNode-class*)*, 116
xmlXIncludes *(*getXIncludes*)*, 43
XMLXIncludeStartNode-class *(*XMLNode-class*)*, 116
xpathApply, *31*, *63*, *70*, *113*, *139*, *143*, *148*
xpathApply *(*getNodeSet*)*, 32
xpathSApply, *66*
xpathSApply *(*getNodeSet*)*, 32