XML: Parsing and Writing

Version 8.15.0.2

Paul Graunke and Jay McCarthy

October 25, 2024

`(require xml)` package: `base`

The `xml` library provides functions for parsing and generating XML. XML can be represented as an instance of the `document` structure type, or as a kind of S-expression that is called an *X-expression*.

The `xml` library does not provide Document Type Declaration (DTD) processing, including preservation of DTDs in read documents, or validation. It also does not expand user-defined entities or read user-defined entities in attributes. It does not interpret namespaces either.

1 Datatypes

```
(struct location (line char offset)
    #:extra-constructor-name make-location)
  line : (or/c #f exact-nonnegative-integer?)
  char : (or/c #f exact-nonnegative-integer?)
  offset : exact-nonnegative-integer?
```

Represents a location in an input stream. The offset is a character offset unless `xml-count-bytes` is `#t`, in which case it is a byte offset.

```
location/c : contract?
```

Equivalent to `(or/c location? symbol? #f)`.

```
(struct source (start stop)
    #:extra-constructor-name make-source)
  start : location/c
  stop : location/c
```

Represents a source location. Other structure types extend `source`.

When XML is generated from an input stream by `read-xml`, locations are represented by `location` instances. When XML structures are generated by `xexpr->xml`, then locations are symbols.

```
(struct external-dtd (system)
    #:extra-constructor-name make-external-dtd)
  system : string?
(struct external-dtd/public external-dtd (public)
    #:extra-constructor-name make-external-dtd/public)
  public : string?
(struct external-dtd/system external-dtd ()
    #:extra-constructor-name make-external-dtd/system)
```

Represents an externally defined DTD.

```
(struct document-type (name external inlined)
    #:extra-constructor-name make-document-type)
  name : symbol?
  external : external-dtd?
  inlined : #f
```

Represents a document type.

```
(struct comment (text)
    #:extra-constructor-name make-comment)
  text : string?
```

Represents a comment.

```
(struct p-i source (target-name instruction)
    #:extra-constructor-name make-p-i)
  target-name : symbol?
  instruction : string?
```

Represents a processing instruction.

```
misc/c : contract?
```

Equivalent to `(or/c comment? p-i?)`

```
(struct prolog (misc dtd misc2)
    #:extra-constructor-name make-prolog)
  misc : (listof misc/c)
  dtd : (or/c document-type #f)
  misc2 : (listof misc/c)
```

Represents a document prolog.

```
(struct document (prolog element misc)
    #:extra-constructor-name make-document)
  prolog : prolog?
  element : element?
  misc : (listof misc/c)
```

Represents a document.

```
(struct element source (name attributes content)
    #:extra-constructor-name make-element)
  name : symbol?
  attributes : (listof attribute?)
  content : (listof content/c)
```

Represents an element.

```
(struct attribute source (name value)
    #:extra-constructor-name make-attribute)
  name : symbol?
  value : (or/c string? permissive/c)
```

Represents an attribute within an element.

```
content/c : contract?
```

Equivalent to `(or/c pcdata? element? entity? comment? cdata? p-i? permissive/c)`.

```
permissive/c : contract?
```

If `(permissive-xexprs)` is `#t`, then equivalent to `any/c`, otherwise equivalent to `(make-none/c 'permissive)`

```
(valid-char? x) → boolean?
  x : any/c
```

Returns true if *x* is an exact-nonnegative-integer whose character interpretation under UTF-8 is from the set ([#x1-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]), in accordance with section 2.2 of the XML 1.1 spec.

```
(struct entity source (text)
    #:extra-constructor-name make-entity)
  text : (or/c symbol? valid-char?)
```

Represents a symbolic or numerical entity.

```
(struct pcdata source (string)
    #:extra-constructor-name make-pcdata)
  string : string?
```

Represents PCDATA content.

```
(struct cdata source (string)
    #:extra-constructor-name make-cdata)
  string : string?
```

Represents CDATA content.

The `string` field is assumed to be of the form `<![CDATA[`⟨*content*⟩`]]>` with proper quoting of ⟨*content*⟩. Otherwise, `write-xml` generates incorrect output.

```
(struct exn:invalid-xexpr exn:fail (code)
    #:extra-constructor-name make-exn:invalid-xexpr)
  code : any/c
```

Raised by `validate-xexpr` when passed an invalid X-expression. The `code` fields contains an invalid part of the input to `validate-xexpr`.

```
(struct exn:xml exn:fail:read ()
    #:extra-constructor-name make-exn:xml)
```

Raised by `read-xml` when an error in the XML input is found.

```
(xexpr? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a X-expression, `#f` otherwise.

The following grammar describes expressions that create X-expressions:

```
xexpr = string
      | (list symbol (list (list symbol string) ...) xexpr ...)
      | (cons symbol (list xexpr ...))
      | symbol
      | valid-char?
      | cdata
      | misc
```

A *string* is literal data. When converted to an XML stream, the characters of the data will be escaped as necessary.

A pair represents an element, optionally with attributes. Each attribute's name is represented by a symbol, and its value is represented by a string.

A *symbol* represents a symbolic entity. For example, `'nbsp` represents ` `.

An `valid-char?` represents a numeric entity. For example, `#x20` represents `&#x20;`.

A *cdata* is an instance of the `cdata` structure type, and a *misc* is an instance of the `comment` or `p-i` structure types.

```
xexpr/c : contract?
```

A contract that is like `xexpr?` except produces a better error message when the value is not an X-expression.

2 X-expression Predicate and Contract

```
(require xml/xexpr)
```
package: base

The xml/xexpr library provides just xexpr/c, xexpr?, correct-xexpr?, and validate-xexpr from xml with minimal dependencies.

3 Reading and Writing XML

```
(read-xml [in]) → document?
  in : input-port? = (current-input-port)
```

Reads in an XML document from the given or current input port. XML documents contain exactly one element, raising `xml-read:error` if the input stream has zero elements or more than one element.

Malformed xml is reported with source locations in the form $\langle l \rangle.\langle c \rangle/\langle o \rangle$, where $\langle l \rangle$, $\langle c \rangle$, and $\langle o \rangle$ are the line number, column number, and next port position, respectively as returned by `port-next-location`.

Any non-characters other than `eof` read from the input-port appear in the document content. Such special values may appear only where XML content may. See `make-input-port` for information about creating ports that return non-character values.

Example:

```
> (xml->xexpr (document-element
               (read-xml (open-input-string
                          "<doc><bold>hi</bold> there!</doc>"))))
'(doc () (bold () "hi") " there!")
```

```
(read-xml/document [in]) → document?
  in : input-port? = (current-input-port)
```

Like `read-xml`, except that the reader stops after the single element, rather than attempting to read "miscellaneous" XML content after the element. The document returned by `read-xml/document` always has an empty `document-misc`.

```
(read-xml/element [in]) → element?
  in : input-port? = (current-input-port)
```

Reads a single XML element from the port. The next non-whitespace character read must start an XML element, but the input port can contain other data after the element.

```
(syntax:read-xml [in #:src source-name]) → syntax?
  in : input-port? = (current-input-port)
  source-name : any/c = (object-name in)
```

Reads in an XML document and produces a syntax object version (like `read-syntax`) of an X-expression.

```
(syntax:read-xml/element [in
                          #:src source-name]) → syntax?
  in : input-port? = (current-input-port)
  source-name : any/c = (object-name in)
```

Like `syntax:real-xml`, but it reads an XML element like `read-xml/element`.

```
(write-xml doc [out]) → void?
  doc : document?
  out : output-port? = (current-output-port)
```

Same as `display-xml` with `#:indentation 'none`.

```
(write-xml/content content [out]) → void?
  content : content/c
  out : output-port? = (current-output-port)
```

Same as `display-xml/content` with `#:indentation 'none`.

```
(display-xml  doc
              [out
               #:indentation indentation]) → void?
  doc : document?
  out : output-port? = (current-output-port)
  indentation : (or/c 'none 'classic 'peek 'scan) = 'classic
```

Writes the document to the given output port.

See `display-xml/content` for an explanation of *indentation*.

```
(display-xml/content  content
                      [out
                       #:indentation indentation]) → void?
  content : content/c
  out : output-port? = (current-output-port)
  indentation : (or/c 'none 'classic 'peek 'scan) = 'classic
```

Writes document content to the given output port.

Indentation can make the output more readable, though less technically correct when whitespace is significant. The four *indentation* modes are as follows:

- `'none` — No whitespace is added. This is the only mode that is guaranteed to be 100% accurate in all situations.

- `'classic` — Whitespace is added around almost every node. This mode is mostly for compatibility.
- `'scan` — If any child of an `element?` is `pcdata?` or `entity?`, then no whitespace will be added inside that element. This mode works well for XML that does not contain mixed content, but `'peek` should be equally good and faster.
- `'peek` — Like `'scan` except only the first child is checked. This mode works well for XML that does not contain mixed content.

Examples:

```
> (define example-data
    '(root (a "nobody")
           (b "some" "body")
           (c "any" (i "body"))
           (d (i "every") "body")))
> (define (show indentation [data example-data])
    (display-xml/content (xexpr->xml data)
                         #:indentation indentation))
;
; `none` is guaranteed to be accurate:
> (show 'none)
<root><a>nobody</a><b>somebody</b><c>any<i>body</i></c><d><i>every</i>body</d></root>
;
; `classic` adds the most whitespace.
; Even the 'nobody' pcdata has whitespace added:
> (show 'classic)

<root>
  <a>
    nobody
  </a>
  <b>
    some
    body
  </b>
  <c>
    any
    <i>
      body
    </i>
  </c>
  <d>
    <i>
      every
```

```
    </i>
    body
  </d>
</root>
;
; `peek` cannot see that <d> contains a pcdata child:
> (show 'peek)

<root>
  <a>nobody</a>
  <b>somebody</b>
  <c>any<i>body</i></c>
  <d>
    <i>every</i>
    body
  </d>
</root>
;
; `scan` sees that <d> contains a pcdata child:
> (show 'scan)

<root>
  <a>nobody</a>
  <b>somebody</b>
  <c>any<i>body</i></c>
  <d><i>every</i>body</d>
</root>
```

Be warned that even `'scan` does not handle HTML with 100% accuracy. The following example will be incorrectly rendered as "*no* **body**" instead of "*no***body**":

Examples:

```
> (define html-data '(span (i "no") (b "body")))
> (show 'scan html-data)

<span>
  <i>no</i>
  <b>body</b>
</span>
```

```
(write-xexpr  xe
              [out
               #:insert-newlines? insert-newlines?]) → void?
  xe : xexpr/c
```

```
out : output-port? = (current-output-port)
insert-newlines? : any/c = #f
```

Writes an X-expression to the given output port, without using an intermediate XML document.

If *insert-newlines?* is true, the X-expression is written with newlines before the closing angle bracket of a tag.

4 XML and X-expression Conversions

```
(permissive-xexprs) → boolean?
(permissive-xexprs v) → void?
  v : any/c
```

If this is set to non-false, then `xml->xexpr` will allow non-XML objects, such as other structs, in the content of the converted XML and leave them in place in the resulting "X-expression".

```
(xml->xexpr content) → xexpr/c
  content : content/c
```

Converts document content into an X-expression, using `permissive-xexprs` to determine if foreign objects are allowed.

```
(xexpr->xml xexpr) → content/c
  xexpr : xexpr/c
```

Converts an X-expression into XML content.

```
(xexpr->string xexpr) → string?
  xexpr : xexpr/c
```

Converts an X-expression into a string containing XML.

```
(string->xexpr str) → xexpr/c
  str : string?
```

Converts XML represented with a string into an X-expression.

```
(xml-attribute-encode str) → string?
  str : string?
```

Escapes a string as required for XML attributes.

The escaping performed for attribute strings is slightly different from that performed for body strings, in that double-quotes must be escaped, as they would otherwise terminate the enclosing string.

Note that this conversion is performed automatically in attribute positions by `xexpr->string`, and you are therefore unlikely to need this function unless you are using `include-template` to insert strings directly into attribute positions of HTML.

Added in version 6.6.0.7 of package `base`.

```
((eliminate-whitespace [tags choose]) elem) → element?
  tags : (listof symbol?) = empty
  choose : (boolean? . -> . boolean?) = (λ (x) x)
  elem : element?
```

Some elements should not contain any text, only other tags, except they often contain whitespace for formatting purposes. Given a list of tag names as `tags` and the identity function as `choose`, `eliminate-whitespace` produces a function that filters out PCDATA consisting solely of whitespace from those elements, and it raises an error if any non-whitespace text appears. Passing in `not` as `choose` filters all elements which are not named in the `tags` list. Using `(lambda (x) #t)` as `choose` filters all elements regardless of the `tags` list.

```
(validate-xexpr v) → #t
  v : any/c
```

If `v` is an X-expression, the result is `#t`. Otherwise, `exn:invalid-xexpr`s is raised, with a message of the form "Expected ⟨*something*⟩, given ⟨*something-else*⟩". The `code` field of the exception is the part of `v` that caused the exception.

Examples:

```
> (validate-xexpr '(doc () "over " (em () "9000") "!"))
#t
> (validate-xexpr #\newline)
Expected a string, symbol, valid numeric entity, comment,
processing instruction, or list, given #\newline
```

```
(correct-xexpr? v success-k fail-k) → any/c
  v : any/c
  success-k : (-> any/c)
  fail-k : (exn:invalid-xexpr? . -> . any/c)
```

Like `validate-xexpr`, except that `success-k` is called on each valid leaf, and `fail-k` is called on invalid leaves; the `fail-k` may return a value instead of raising an exception or otherwise escaping. Results from the leaves are combined with `and` to arrive at the final result.

5 Parameters

```
(current-unescaped-tags) → (listof symbol?)
(current-unescaped-tags tags) → void?
  tags : (listof symbol?)
 = '()
```

A parameter that determines which tags' string contents should not be escaped. For backwards compatibility, this defaults to the empty list.

Added in version 8.0.0.12 of package `base`.

```
html-unescaped-tags : (listof symbol?) = '(script style)
```

The list of tags whose contents are normally not escaped in HTML. See `current-unescaped-tags`.

Example:

```
> (parameterize ([current-unescaped-tags html-unescaped-tags])
    (write-xexpr '(html
                   (p "1 < 2")
                   (script "1 < 2"))))
<html><p>1 &lt; 2</p><script>1 < 2</script></html>
```

Added in version 8.0.0.12 of package `base`.

```
(empty-tag-shorthand) → (or/c 'always 'never (listof symbol?))
(empty-tag-shorthand shorthand) → void?
  shorthand : (or/c 'always 'never (listof symbol?))
```

A parameter that determines whether output functions should use the <⟨*tag*⟩/> tag notation instead of <⟨*tag*⟩></⟨*tag*⟩> for elements that have no content.

When the parameter is set to `'always`, the abbreviated notation is always used. When set of `'never`, the abbreviated notation is never generated. when set to a list of symbols is provided, tags with names in the list are abbreviated.

The abbreviated form is the preferred XML notation. However, most browsers designed for HTML will only properly render XHTML if the document uses a mixture of the two formats. The `html-empty-tags` constant contains the W3 consortium's recommended list of XHTML tags that should use the shorthand. This list is the default value of `empty-tag-shorthand`.

```
html-empty-tags : (listof symbol?)
 = '(param meta link isindex input img hr frame col br basefont base area)
```

See empty-tag-shorthand.

Example:

```
> (parameterize ([empty-tag-shorthand html-empty-tags])
    (write-xml/content (xexpr->xml `(html
                                     (body ((bgcolor "red"))
                                       "Hi!" (br) "Bye!")))))
<html><body bgcolor="red">Hi!<br />Bye!</body></html>
```

```
(collapse-whitespace) → boolean?
(collapse-whitespace collapse?) → void?
  collapse? : any/c
```

A parameter that controls whether consecutive whitespace is replaced by a single space. CDATA sections are not affected. The default is #f.

```
(read-comments) → boolean?
(read-comments preserve?) → void?
  preserve? : any/c
```

A parameter that determines whether comments are preserved or discarded when reading XML. The default is #f, which discards comments.

```
(xml-count-bytes) → boolean?
(xml-count-bytes count-bytes?) → void?
  count-bytes? : any/c
```

A parameter that determines whether read-xml counts characters or bytes in its location tracking. The default is #f, which counts characters.

You may want to use #t if, for example, you will be communicating these offsets to a C program that can more easily deal with byte offsets into the character stream, as opposed to UTF-8 character offsets.

```
(xexpr-drop-empty-attributes) → boolean?
(xexpr-drop-empty-attributes drop?) → void?
  drop? : any/c
```

Controls whether xml->xexpr drops or preserves attribute sections for an element that has no attributes. The default is #f, which means that all generated X-expression elements have an attributes list (even if it's empty).

6 PList Library

```
(require xml/plist)
```
package: base

The xml/plist library provides the ability to read and write XML documents that conform to the *plist* DTD, which is used to store dictionaries of string–value associations. This format is used by Mac OS (both the operating system and its applications) to store all kinds of data.

A *plist value* is a value that could be created by an expression matching the following *pl-expr* grammar, where a value created by a *dict-expr* is a *plist dictionary*:

```
    pl-expr = string
            | (list 'true)
            | (list 'false)
            | (list 'integer integer)
            | (list 'real real)
            | (list 'data string)
            | (list 'date string)
            | dict-expr
            | (list 'array pl-expr ...)

  dict-expr = (list 'dict assoc-pair ...)

 assoc-pair = (list 'assoc-pair string pl-expr)
```

```
(plist-value? any/c) → boolean?
  any/c : v
```

Returns #t if v is a plist value, #f otherwise.

```
(plist-dict? any/c) → boolean?
  any/c : v
```

Returns #t if v is a plist dictionary, #f otherwise.

```
(read-plist in) → plist-value?
  in : input-port?
```

Reads a plist from a port, and produces a plist value.

```
(write-plist dict out) → void?
  dict : plist-value?
  out : output-port?
```

Write a plist value to the given port.

Examples:

```
> (define my-dict
    `(dict (assoc-pair "first-key"
                       "just a string with some  whitespace")
           (assoc-pair "second-key"
                       (false))
           (assoc-pair "third-key"
                       (dict))
           (assoc-pair "fourth-key"
                       (dict (assoc-pair "inner-key"
                                         (real 3.432))))
           (assoc-pair "fifth-key"
                       (array (integer 14)
                              "another string"
                              (true)))
           (assoc-pair "sixth-key"
                       (array))
           (assoc-pair "seventh-key"
                       (data "some data"))
           (assoc-pair "eighth-key"
                       (date "2013-05-10T20:29:55Z"))))
> (define-values (in out) (make-pipe))
> (write-plist my-dict out)
> (close-output-port out)
> (define new-dict (read-plist in))
> (equal? my-dict new-dict)
#t
```

The XML generated by `write-plist` in the above example looks like the following, if re-formatted by hand to have newlines and indentation:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM
 "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
  <dict>
    <key>first-key</key>
    <string>just a string with some  whitespace</string>
    <key>second-key</key>
    <false />
    <key>third-key</key>
    <dict />
    <key>fourth-key</key>
    <dict>
      <key>inner-key</key>
      <real>3.432</real>
```

```
    </dict>
    <key>fifth-key</key>
    <array>
      <integer>14</integer>
      <string>another string</string>
      <true />
    </array>
    <key>sixth-key</key>
    <array />
    <key>seventh-key</key>
    <data>some data</data>
    <key>eighth-key</key>
    <date>2013-05-10T20:29:55Z</date>
  </dict>
</plist>
```

7 Simple X-expression Path Queries

```
(require xml/path)
```
package: base

This library provides a simple path query library for X-expressions.

```
se-path? : contract?
```

A sequence of symbols followed by an optional keyword.

The prefix of symbols specifies a path of tags from the leaves with an implicit any sequence to the root. The final, optional keyword specifies an attribute.

```
(se-path*/list p xe) → (listof any/c)
  p : se-path?
  xe : xexpr?
```

Returns a list of all values specified by the path *p* in the X-expression *xe*.

```
(se-path* p xe) → any/c
  p : se-path?
  xe : xexpr?
```

Returns the first answer from `(se-path*/list p xe)`.

Examples:

```
> (define some-page
    '(html (body (p ([class "awesome"]) "Hey") (p "Bar"))))
> (se-path*/list '(p) some-page)
'("Hey" "Bar")
> (se-path* '(p) some-page)
"Hey"
> (se-path* '(p #:class) some-page)
"awesome"
> (se-path*/list '(body) some-page)
'((p ((class "awesome")) "Hey") (p "Bar"))
> (se-path*/list '() some-page)
'((html (body (p ((class "awesome")) "Hey") (p "Bar")))
  (body (p ((class "awesome")) "Hey") (p "Bar"))
  (p ((class "awesome")) "Hey")
  "Hey"
  (p "Bar")
  "Bar")
```