

# XML: Parsing and Writing

Version 4.1.5

Paul Graunke and Jay McCarthy

March 21, 2009

`(require xml)`

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 interpret namespaces either.

# 1 Datatypes

---

```
(struct location (line char offset))
  line : exact-nonnegative-integer?
  char : exact-nonnegative-integer?
  offset : exact-nonnegative-integer?
```

Represents a location in an input stream.

---

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

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

---

```
(struct source (start stop))
  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))
  system : string?
(struct (external-dtd/public external-dtd) (public))
  public : string?
(struct (external-dtd/system external-dtd) ())
```

Represents an externally defined DTD.

---

```
(struct document-type (name external inlined))
  name : symbol?
  external : external-dtd?
  inlined : false/c
```

Represents a document type.

---

```
(struct comment (text))
  text : string?
```

Represents a comment.

---

```
(struct (p-i source) (target-name instruction))
  target-name : symbol?
  instruction : string?
```

Represents a processing instruction.

---

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

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

---

```
(struct prolog (misc dtd misc2))
  misc : (listof misc/c)
  dtd : (or/c document-type false/c)
  misc2 : (listof misc/c)
```

Represents a document prolog.

---

```
(struct document (prolog element misc))
  prolog : prolog?
  element : element?
  misc : (listof misc/c)
```

Represents a document.

---

```
(struct (element source) (name attributes content))
  name : symbol?
  attributes : (listof attribute?)
  content : (listof content/c)
```

Represents an element.

---

```
(struct (attribute source) (name value))
  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?)` is `#t`, then equivalent to `any/c`, otherwise equivalent to `(make-none/c 'permissive)`

---

```
(struct (entity source) (text))
  text : (or/c symbol? exact-nonnegative-integer?)
```

Represents a symbolic or numerical entity.

---

```
(struct (pcdata source) (string))
  string : string?
```

Represents PCDATA content.

---

```
(struct (cdata source) (string))
  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))
  code : any/c
```

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

---

```
(struct (exn:xml exn:fail:read) ())
```

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  
| exact-nonnegative-integer  
| 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 `&nbsp;`.

An *exact-nonnegative-integer* represents a numeric entity. For example, `#x20` represents `&#20;`.

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 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.

Examples:

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

---

```
(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]) → syntax?  
in : input-port? = (current-input-port)
```

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

---

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

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

---

```
(write-xml doc [out]) → void?  
doc : document?
```

```
out : output-port? = (current-output-port)
```

Writes a document to the given output port, currently ignoring everything except the document's root element.

---

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

Writes document content to the given output port.

---

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

Like `write-xml`, but newlines and indentation make the output more readable, though less technically correct when whitespace is significant.

---

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

Like `write-xml/content`, but with indentation and newlines like `display-xml`.

### 3 XML and X-expression Conversions

---

```
(permissive?) → boolean?  
(permissive? 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?` 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.

---

```
((eliminate-whitespace tags choose) elem) → element?  
tags : (listof symbol?)  
choose : (boolean? . -> . boolean?)  
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) → (one-of/c #t)  
v : any/c
```

If *v* is an X-expression, the result `#t`. Otherwise, `exn:invalid-exprs` is raised, with the 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.

---

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

Like `validate-expr`, 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 of otherwise escaping. Results from the leaves are combined with `and` to arrive at the final result.

## 4 Parameters

---

```
(empty-tag-shorthand)
→ (or/c (one-of/c 'always 'never) (listof symbol?))
(empty-tag-shorthand shorthand) → void?
shorthand : (or/c (one-of/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 to `'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 default is `'always`.

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.

---

```
html-empty-tags : (listof symbol?)
```

See `empty-tag-shorthand`.

Examples:

```
> (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.

---

```
(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).

## 5 PList Library

```
(require xml/plist)
```

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 X (both the operating system and its applications) to store all kinds of data.

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

```
dict-expr = (list (quote dict) assoc-pair ...)  
  
assoc-pair = (list (quote assoc-pair) string pl-value)  
  
pl-value = string  
          | (list (quote true))  
          | (list (quote false))  
          | (list (quote integer) integer)  
          | (list (quote real) real)  
          | dict-expr  
          | (list (quote array) pl-value ...)
```

---

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

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

---

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

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

---

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

Write a plist dictionary 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))))
> (define-values (in out) (make-pipe))
> (write-plist my-dict out)
(file "/var/tmp/pli/collects/xml/plist.ss") broke the
contract
(-> symbol? external-dtd? #f document-type?)
on make-document-type; expected <external-dtd?>, given: #f
> (close-output-port out)
> (define new-dict (read-plist in))
read-xml: parse-error: expected root element - received
#<eof>
> (equal? my-dict new-dict)
reference to undefined identifier: new-dict

```

The XML generated by `write-plist` in the above example looks like the following, if re-formatted by:

```

<?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 />
</dict>
</plist>
```