XML: Parsing and Writing

Version 5.2

Paul Graunke and Jay McCarthy

November 8, 2011

```
(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 not interpret namespaces either.

1 Datatypes

```
(struct location (line char offset)
        #:extra-constructor-name make-location)
  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)
        #: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 : false/c
```

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 false/c)
  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 (#x9 | #xA | #xD | [#x20-#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 `&#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.

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]) → 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: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)
```

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`.

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

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

3 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.

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

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.

```
(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 'dict assoc-pair ...)

 assoc-pair = (list 'assoc-pair string pl-value)

  pl-value = string
           | (list 'true)
           | (list 'false)
           | (list 'integer integer)
           | (list 'real real)
           | dict-expr
           | (list '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)
> (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:

```
<?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>
```

6 Simple X-expression Path Queries

```
(require xml/path)
```

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")
```