



xmerl

Copyright © 2004-2019 Ericsson AB. All Rights Reserved.  
xmerl 1.3.21  
July 11, 2019

---

**Copyright © 2004-2019 Ericsson AB. All Rights Reserved.**

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0> Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

**July 11, 2019**





```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE motorcycles SYSTEM "motorcycles.dtd">
<motorcycles>
  <bike year="2000" color="black">
    <name>
      <manufacturer>Suzuki</manufacturer>
      <brandName>Suzuki VL 1500</brandName>
      <additionalName>Intruder</additionalName>
    </name>
    <engine>V-engine, 2-cylinders, 1500 cc</engine>
    <kind>custom</kind>
    <drive>cardan</drive>
    <accessories>Sissy bar, luggage carrier,V&H exhaust pipes</accessories>
  </bike>
  <date>2004.08.25</date>
  <bike year="1983" color="read pearl">
    <name>
      <manufacturer>Yamaha</manufacturer>
      <brandName>XJ 400</brandName>
    </name>
    <engine>4 cylinder, 400 cc</engine>
    <kind>alround</kind>
    <drive>chain</drive>
    <comment>Good shape!</comment>
  </bike>
</motorcycles>
```

and motorcycles.dtd looks like:

```
<?xml version="1.0" encoding="utf-8" ?>
<ELEMENT motorcycles (bike,date?)+ >
<ELEMENT bike (name,engine,kind,drive, accessories?,comment?) >
<ELEMENT name (manufacturer,brandName,additionalName?) >
<ELEMENT manufacturer (#PCDATA)>
<ELEMENT brandName (#PCDATA)>
<ELEMENT additionalName (#PCDATA)>
<ELEMENT engine (#PCDATA)>
<ELEMENT kind (#PCDATA)>
<ELEMENT drive (#PCDATA)>
<ELEMENT comment (#PCDATA)>
<ELEMENT accessories (#PCDATA)>

<!-- Date of the format yyyy.mm.dd -->
<ELEMENT date (#PCDATA)>
<ATTLIST bike year NMTOKEN #REQUIRED
            color NMTOKENS #REQUIRED
            condition (useless | bad | serviceable | moderate | good |
                      excellent | new | outstanding) "excellent" >
```

If you want to parse the XML file motorcycles.xml you run it in the Erlang shell like:





## 1.1 xmerl

---

```
{RootEl,Misc}=xmerl_scan:file('motorcycles.xml'),
#xmlElement{content=Content} = RootEl,
NewContent=Content++lists:flatten([Data]),
NewRootEl=RootEl#xmlElement{content=NewContent},
```

Then you can run it through the `export_simple/2` function:

```
{ok,IOF}=file:open('new_motorcycles.xml',[write]),
Export=xmerl:export_simple([NewRootEl],xmerl_xml),
io:format(IOF,"~s~n",[lists:flatten(Export)]),
```

The result would be:

```
<?xml version="1.0"?><motorcycles>
  <bike year="2000" color="black">
    <name>
      <manufacturer>Suzuki</manufacturer>
      <brandName>Suzuki VL 1500</brandName>
      <additionalName>Intruder</additionalName>
    </name>
    <engine>V-engine, 2-cylinders, 1500 cc</engine>
    <kind>custom</kind>
    <drive>cardan</drive>
    <accessories>Sissy bar, luggage carrier,V&H exhaust pipes</accessories>
  </bike>
  <date>2004.08.25</date>
  <bike year="1983" color="read pearl">
    <name>
      <manufacturer>Yamaha</manufacturer>
      <brandName>XJ 400</brandName>
    </name>
    <engine>4 cylinder, 400 cc</engine>
    <kind>alround</kind>
    <drive>chain</drive>
    <comment>Good shape!</comment>
  </bike>
  <bike year="2003" color="black" condition="new"><name><manufacturer>Harley Davidsson</manufacturer><brandName>
```

If it is important to get similar indentation and newlines as in the original document you have to add `#xmlText{}` records with space and newline values in appropriate places. It may also be necessary to keep the original prolog where the DTD is referenced. If so, it is possible to pass a `RootAttribute {prolog,Value}` to `export_simple/3`. The following example code fixes those changes in the previous example:

```
Data =
  [#xmlText{value=" "},
   {bike,[{year,"2003"},{color,"black"},{condition,"new"}],
    [#xmlText{value="\
"},
     {name,[#xmlText{value="\
"},
      {manufacturer,["Harley Davidsson"]},
      #xmlText{value="\
"},
      {brandName,["XL1200C"]},
      #xmlText{value="\
"},
      {additionalName,["Sportster"]},
      #xmlText{value="\
"}]]},
    {engine,["V-engine, 2-cylinders, 1200 cc"]},
    #xmlText{value="\
"},
    {kind,["custom"]},
    #xmlText{value="\
"},
    {drive,["belt"]},
    #xmlText{value="\
"}]]},
  #xmlText{value="\
"}],
...
NewContent=Content++lists:flatten([Data]),
NewRootEl=RootEl#xmlElement{content=NewContent},
...
Prolog = ["<?xml version=\\\"1.0\\\" encoding=\\\"utf-8\\\" ?>
<!DOCTYPE motorcycles SYSTEM \\\"motorcycles.dtd\\\">\\
"],
Export=xmerl:export_simple([NewRootEl],xmerl_xml,[{prolog,Prolog}]),
...
```

The result will be:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE motorcycles SYSTEM "motorcycles.dtd">
<motorcycles>
  <bike year="2000" color="black">
    <name>
      <manufacturer>Suzuki</manufacturer>
      <brandName>Suzuki VL 1500</brandName>
      <additionalName>Intruder</additionalName>
    </name>
    <engine>V-engine, 2-cylinders, 1500 cc</engine>
    <kind>custom</kind>
    <drive>cardan</drive>
    <accessories>Sissy bar, luggage carrier,V&H exhaust pipes</accessories>
  </bike>
  <date>2004.08.25</date>
  <bike year="1983" color="read pearl">
    <name>
      <manufacturer>Yamaha</manufacturer>
      <brandName>XJ 400</brandName>
    </name>
    <engine>4 cylinder, 400 cc</engine>
    <kind>alround</kind>
    <drive>chain</drive>
    <comment>Good shape!</comment>
  </bike>
  <bike year="2003" color="black" condition="new">
    <name>
      <manufacturer>Harley Davidsson</manufacturer>
      <brandName>XL1200C</brandName>
      <additionalName>Sportster</additionalName>
    </name><engine>V-engine, 2-cylinders, 1200 cc</engine>
    <kind>custom</kind>
    <drive>belt</drive>
  </bike>
</motorcycles>
```

### 1.1.6 Example: Transforming XML To HTML

Assume that you want to transform the *motorcycles.xml* document to HTML. If you want the same structure and tags of the resulting HTML document as of the XML document then you can use the `xmerl:export/2` function. The following:







```
[["<a name=\"",Name,"\"></>"],V|F(Rest,Name,F)];
([],_,_) -> []
end,
    SortedRefed=InsertRefName_UnWrap(SortedTS,no_name,InsertRefName_UnWrap),
%   SortedTs=[Y||{X,Y}<-lists:keysort(1,Tuples)],
    WS = "\n",
    Fun=fun([H|T],Acc,F)->
    F(T,[H,WS|Acc],F);
    ([],Acc,_F)->
    lists:reverse([WS|Acc])
end,
    if length(SortedRefed) > 0 ->
        Fun(SortedRefed,[],Fun);
        true -> []
    end.

%% removes all but the first of an element in L and inserts a html
%% reference for each list element.
remove_duplicates(L) ->
    remove_duplicates(L,[]).

remove_duplicates([],Acc) ->
    make_ref(lists:sort(lists:reverse(Acc)));
remove_duplicates([A|L],Acc) ->
    case lists:delete(A,L) of
    L ->
        remove_duplicates(L,[A|Acc]);
    L1 ->
        remove_duplicates([A|L1],[Acc])
    end.

make_ref([]) -> [];
make_ref([H]) when is_atom(H) ->
    "<ul><a href=\"#"++atom_to_list(H)+"\">"+atom_to_list(H)+"</a></ul>";
make_ref([H]) when is_list(H) ->
    "<ul><a href=\"#"++H++ "\">\s"++H++"</a></ul>";
make_ref([H|T]) when is_atom(H) ->
    ["<ul><a href=\"#"++atom_to_list(H)+"\">\s"++atom_to_list(H)+"",\n</a></ul>"
    |make_ref(T)];
make_ref([H|T]) when is_list(H) ->
    ["<ul><a href=\"#"++H++ "\">\s"++H++",\n</a></ul>"|make_ref(T)].
```

If we run it like this: `motorcycles2html:process_to_file('result_xs.html', 'motorcycles2.xml')`. The result will be **result\_xs.html**. When the input file is of the same structure as the previous "motorcycles" XML files but it has a little more 'bike' elements and the 'manufacturer' elements are not in order.

## 2 Reference Manual

---

The **xmerl** application contains modules with support for processing of xml files compliant to XML 1.0.







```
rules_state(S::global_state()) -> global_state()
```

Equivalent to *rules\_state(RulesState, S)*.

```
rules_state(X::RulesState, S::global_state()) -> global_state()
```

For controlling the RulesState, to be used in a rules function, and called when the parser store scanner information in a rules database. See **tutorial** on customization functions.

```
string(Text::list()) -> {xmlElement(), Rest}
```

Types:

```
Rest = list()
```

Equivalent to *string(Text, [])*.

```
string(Text::list(), Options::option_list()) -> {document(), Rest}
```

Types:

```
Rest = list()
```

Parse string containing an XML document

```
user_state(S::global_state()) -> global_state()
```

Equivalent to *user\_state(UserState, S)*.

```
user_state(X::UserState, S::global_state()) -> global_state()
```

For controlling the UserState, to be used in a user function. See **tutorial** on customization functions.





RootAttributes is a list of:

- `XmlAttributes = #XmlAttribute{}`

See `export/3` for details on the callback module and the root attributes. The XML-data is always converted to normal form before being passed to the callback module.

**See also:** `export/3`, `export_simple/2`.

`export_simple_content(Content, Callback) -> term()`

Exports simple XML content directly, without further context.

`export_simple_element(Content, Callback) -> term()`

Exports a simple XML element directly, without further context.



```
xslapply(Fun::Function, EList::list()) -> List
```

Types:

```
Function = () -> list()
```

xslapply is a wrapper to make things look similar to xsl:apply-templates.

Example, original XSLT:

```
<xsl:template match="doc/title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>
```

becomes in Erlang:

```
template(E = #xmlElement{ parents=[{'doc',_}|_], name='title'}) ->
  ["<h1>",
   xslapply(fun template/1, E),
   "</h1>"];
```





```
Fname = string()  
CallbackModule = atom()  
Options = option_list()
```

Parse file containing an XML document as a stream, SAX style. Wrapper for a call to the XML parser `xmerl_scan` with a `continuation_fun` for handling streams of XML data. Note that the `continuation_fun`, `acc_fun`, `fetch_fun`, `rules`, `hook_fun`, `close_fun` and `user_state` options cannot be user defined using this parser.

```
string_sax(String::list(), CallbackModule::atom(), UserState,  
Options::option_list()) -> xmlElement()
```

Parse file containing an XML document, SAX style. Wrapper for a call to the XML parser `xmerl_scan` with a `hook_fun` for using `xmerl` export functionality directly after an entity is parsed.



```
string(Str, Doc, Options) -> [docEntity()] | Scalar
```

Equivalent to *string(Str, Doc, [], Doc, Options)*.

```
string(Str, Node, Parents, Doc, Options) -> [docEntity()] | Scalar
```

Types:

```
Str = xPathString()  
Node = nodeEntity()  
Parents = parentList()  
Doc = nodeEntity()  
Options = option_list()  
Scalar = #xmlObj{}
```

Extracts the nodes from the parsed XML tree according to XPath. xmlObj is a record with fields type and value, where type is boolean | number | string









`{endPrefixMapping, Prefix}`  
End the scope of a prefix-URI mapping.

- `Prefix = string()`

`{startElement, Uri, LocalName, QualifiedName, Attributes}`  
Receive notification of the beginning of an element. The Parser will send this event at the beginning of every element in the XML document; there will be a corresponding `endElement` event for every `startElement` event (even when the element is empty). All of the element's content will be reported, in order, before the corresponding `endElement` event.

- `Uri = string()`
- `LocalName = string()`
- `QualifiedName = {Prefix, LocalName}`
- `Prefix = string()`
- `Attributes = [{Uri, Prefix, AttributeName, Value}]`
- `AttributeName = string()`
- `Value = string()`

`{endElement, Uri, LocalName, QualifiedName}`  
Receive notification of the end of an element. The SAX parser will send this event at the end of every element in the XML document; there will be a corresponding `startElement` event for every `endElement` event (even when the element is empty).

- `Uri = string()`
- `LocalName = string()`
- `QualifiedName = {Prefix, LocalName}`
- `Prefix = string()`

`{characters, string()}`  
Receive notification of character data.

`{ignorableWhitespace, string()}`  
Receive notification of ignorable whitespace in element content.

`{processingInstruction, Target, Data}`  
Receive notification of a processing instruction. The Parser will send this event once for each processing instruction found: note that processing instructions may occur before or after the main document element.

- `Target = string()`
- `Data = string()`

`{comment, string()}`  
Report an XML comment anywhere in the document (both inside and outside of the document element).

`startCDATA`  
Report the start of a CDATA section. The contents of the CDATA section will be reported through the regular characters event.

`endCDATA`  
Report the end of a CDATA section.

`{startDTD, Name, PublicId, SystemId}`  
Report the start of DTD declarations, it's reporting the start of the DOCTYPE declaration. If the document has no DOCTYPE declaration, this event will not be sent.

- `Name = string()`
- `PublicId = string()`
- `SystemId = string()`

`endDTD`  
Report the end of DTD declarations, it's reporting the end of the DOCTYPE declaration.

`{startEntity, SysId}`  
Report the beginning of some internal and external XML entities. ???

`{endEntity, SysId}`  
Report the end of an entity. ???

`{elementDecl, Name, Model}`  
Report an element type declaration. The content model will consist of the string "EMPTY", the string "ANY", or a parenthesised group, optionally followed by an occurrence indicator. The model will be normalized so that all parameter entities are fully resolved and all whitespace is removed, and will include the enclosing parentheses. Other normalization (such as removing redundant parentheses or simplifying occurrence indicators) is at the discretion of the parser.

- `Name = string()`
- `Model = string()`

`{attributeDecl, ElementName, AttributeName, Type, Mode, Value}`  
Report an attribute type declaration.

- `ElementName = string()`
- `AttributeName = string()`
- `Type = string()`
- `Mode = string()`
- `Value = string()`

`{internalEntityDecl, Name, Value}`  
Report an internal entity declaration.

- `Name = string()`
- `Value = string()`

`{externalEntityDecl, Name, PublicId, SystemId}`  
Report a parsed external entity declaration.

- `Name = string()`
- `PublicId = string()`
- `SystemId = string()`

`{unparsedEntityDecl, Name, PublicId, SystemId, Ndata}`  
Receive notification of an unparsed entity declaration event.

- `Name = string()`
- `PublicId = string()`
- `SystemId = string()`
- `Ndata = string()`

`{notationDecl, Name, PublicId, SystemId}`  
Receive notification of a notation declaration event.

- `Name = string()`
- `PublicId = string()`
- `SystemId = string()`

`unicode_char()`  
Integer representing valid unicode codepoint.

`unicode_binary()`  
Binary with characters encoded in UTF-8 or UTF-16.

`latin1_binary()`  
Binary with characters encoded in iso-latin-1.

## Exports

`file(Filename, Options) -> Result`

Types:

```
Filename = string()
Options = [option()]
Result = {ok, EventState, Rest} |
  {Tag, Location, Reason, EndTags, EventState}
Rest = unicode_binary() | latin1_binary()
Tag = atom() (fatal_error, or user defined tag)
Location = {CurrentLocation, EntityName, LineNo}
CurrentLocation = string()
EntityName = string()
LineNo = integer()
EventState = term()
Reason = term()
```

Parse file containing an XML document. This functions uses a default continuation function to read the file in blocks.

`stream(Xml, Options) -> Result`

Types:

```
Xml = unicode_binary() | latin1_binary() | [unicode_char()]
Options = [option()]
Result = {ok, EventState, Rest} |
  {Tag, Location, Reason, EndTags, EventState}
Rest = unicode_binary() | latin1_binary() | [unicode_char()]
Tag = atom() (fatal_error or user defined tag)
Location = {CurrentLocation, EntityName, LineNo}
CurrentLocation = string()
EntityName = string()
LineNo = integer()
EventState = term()
Reason = term()
```

Parse a stream containing an XML document.

## CALLBACK FUNCTIONS

The callback interface is based on that the user sends a fun with the correct signature to the parser.

## Exports

`ContinuationFun(State) -> {NewBytes, NewState}`

Types:

```
State = NewState = term()
NewBytes = binary() | list() (should be same as start input in stream/2)
```

This function is called whenever the parser runs out of input data. If the function can't get hold of more input an empty list or binary (depends on start input in stream/2) is returned. Other types of errors is handled through exceptions. Use throw/1 to send the following tuple {Tag = atom(), Reason = string()} if the continuation function encounters a fatal error. Tag is an atom that identifies the functional entity that sends the exception and Reason is a string that describes the problem.

**EventFun(Event, Location, State) -> NewState**

Types:

```
Event = event()  
Location = {CurrentLocation, Entityname, LineNo}  
CurrentLocation = string()  
Entityname = string()  
LineNo = integer()  
State = NewState = term()
```

This function is called for every event sent by the parser. The error handling is done through exceptions. Use throw/1 to send the following tuple {Tag = atom(), Reason = string()} if the application encounters a fatal error. Tag is an atom that identifies the functional entity that sends the exception and Reason is a string that describes the problem.