

xmerl

Copyright © 2004-2026 Ericsson AB. All Rights Reserved.
xmerl 1.3.34.3
November 23, 2026

Copyright © 2004-2026 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..

November 23, 2026

1 xmerl User's Guide

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

1.1 xmerl

1.1.1 Introduction

Features

The **xmerl** XML parser is able to parse XML documents according to the XML 1.0 standard. As default it performs well-formed parsing, (syntax checks and checks of well-formed constraints). Optionally one can also use xmerl as a validating parser, (validate according to referenced DTD and validating constraints). By means of for example the xmerl_xs module it is possible to transform the parsed result to other formats, e.g. text, HTML, XML etc.

Overview

This document does not give an introduction to XML. There are a lot of books available that describe XML from different views. At the **www.W3.org** site you will find the **XML 1.0 specification** and other related specs. One site were you can find tutorials on XML and related specs is **ZVON.org**.

However, here you will find some examples of how to use and to what you can use xmerl. A detailed description of the user interface can be found in the reference manual.

There are two known shortcomings in xmerl:

- It cannot retrieve external entities on the Internet by a URL reference, only resources in the local file system.
- xmerl can parse Unicode encoded data. But, it fails on tag names, attribute names and other mark-up names that are encoded Unicode characters not mapping on ASCII.

By parsing an XML document you will get a record, displaying the structure of the document, as return value. The record also holds the data of the document. xmerl is convenient to use in for instance the following scenarios:

You need to retrieve data from XML documents. Your Erlang software can handle information from the XML document by extracting data from the data structure received by parsing.

It is also possible to do further processing of parsed XML with xmerl. If you want to change format of the XML document to for instance HTML, text or other XML format you can transform it. There is support for such transformations in xmerl.

One may also convert arbitrary data to XML. So it for instance is easy to make it readable by humans. In this case you first create xmerl data structures out of your data, then transform it to XML.

You can find examples of these three examples of usage below.

1.1.2 xmerl User Interface Data Structure

The following records used by xmerl to save the parsed data are defined in `xmerl.hrl`

The result of a successful parsing is a tuple `{DataStructure,M}`. `M` is the XML production Misc, which is the mark-up that comes after the element of the document. It is returned "as is". `DataStructure` is an `xmlElement` record, that among others have the fields `name`, `parents`, `attributes` and `content` like:

```
#xmlElement{name=Name,
            ...
            parents=Parents,
            ...
            attributes=Attrs,
            content=Content,
            ...}
```

The name of the element is found in the `name` field. In the `parents` field is the names of the parent elements saved. Parents is a list of tuples where the first element in each tuple is the name of the parent element. The list is in reverse order.

The record `xmlAttribute` holds the name and value of an attribute in the fields `name` and `value`. All attributes of an element is a list of xmlAttribute in the field `attributes` of the xmlElement record.

The `content` field of the top element is a list of records that shows the structure and data of the document. If it is a simple document like:

```
<?xml version="1.0"?>
<dog>
Grand Danois
</dog>
```

The parse result will be:

```
#xmlElement{name = dog,
            ...
            parents = [],
            ...
            attributes = [],
            content = [{xmlText,[{dog,1}],1,[],"\
Grand Danois\
",text}],
            ...
            }
```

Where the content of the top element is: `[{xmlText,[{dog,1}],1,[],"\ Grand Danois\ ",text}]`. Text will be returned in `xmlText` records. Though, usually documents are more complex, and the content of the top element will in that case be a nested structure with xmlElement records that in turn may have complex content. All of this reflects the structure of the XML document.

Space characters between mark-up as `space`, `tab` and `line feed` are normalized and returned as xmlText records.

Errors

An unsuccessful parse results in an error, which may be a tuple `{error,Reason}` or an exit: `{'EXIT',Reason}`. According to the XML 1.0 standard there are `fatal error` and `error` situations. The fatal errors **must** be detected by a conforming parser while an error **may** be detected. Both categories of errors are reported as fatal errors by this version of xmerl, most often as an exit.

1.1.3 Getting Started

In the following examples we use the XML file "motorcycles.xml" and the corresponding DTD "motorcycles.dtd". motorcycles.xml looks like:

```
<?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&amp;amp;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:

```
3> {ParseResult,Misc}=xmerl_scan:file("motorcycles.xml").
{{xmlElement,motorcycles,
             motorcycles,
             [],
             {xmlNamespace,[],[]},
             [],
             1,
             [],
             [{xmlText,[{motorcycles,1}],1,[],"\
  ",text},
              {xmlElement,bike,
                          bike,
                          [],
                          {xmlNamespace,[],[]},
                          [{motorcycles,1}],
                          2,
                          [{xmlAttribute,year,[],[],[],[]|...},
                           {xmlAttribute,color,[],[],[]|...}],
                          [{xmlText,[{bike,2},{motorcycles|...}],
                                    1,
                                    []|...},
                           {xmlElement,name,name,[]|...},
                           {xmlText,[{...}|...],3|...},
                           {xmlElement,engine|...},
                           {xmlText|...},
                           {...}|...],
                          [],
                          ".",
                          undeclared},
              ...
              ],
             [],
             ".",
             undeclared},
 []}
4>
```

If you instead receives the XML doc as a string you can parse it by `xmerl_scan:string/1`. Both file/2 and string/2 exists where the second argument is a list of options to the parser, see the reference manual.

1.1.4 Example: Extracting Data From XML Content

In this example consider the situation where you want to examine a particular data in the XML file. For instance, you want to check for how long each motorcycle have been recorded.

Take a look at the DTD and observe that the structure of an XML document that is conformant to this DTD must have one motorcycles element (the root element). The motorcycles element must have at least one bike element. After each bike element it may be a date element. The content of the date element is #PCDATA (Parsed Character DATA), i.e. raw text. Observe that if #PCDATA must have a `"<"` or a `"&"` character it must be written as `"<"` and `"&"` respectively. Also other character entities exists similar to the ones in HTML and SGML.

If you successfully parse the XML file with the validation on as in: `xmerl_scan:file('motorcycles.xml', [{validation,true}])` you know that the XML document is valid and has the structure according to the DTD.

Thus, knowing the allowed structure it is easy to write a program that traverses the data structure and picks the information in the xmlElements records with name date.

Observe that white space: each space, tab or line feed, between mark-up results in an xmlText record.

1.1.5 Example: Create XML Out Of Arbitrary Data

For this task there are more than one way to go. The "brute force" method is to create the records you need and feed your data in the content and attribute fields of the appropriate element.

There is support for this in xmerl by the "simple-form" format. You can put your data in a simple-form data structure and feed it into `xmerl:export_simple(Content,Callback,RootAttributes)`. Content may be a mixture of simple-form and xmerl records as xmlElement and xmlText.

The Types are:

- Content = [Element]
- Callback = atom()
- RootAttributes = [Attributes]

Element is any of:

- {Tag, Attributes, Content}
- {Tag, Content}
- Tag
- IOString
- #xmlText{}
- #xmlElement{}
- #xmlPI{}
- #xmlComment{}
- #xmlDecl{}

The simple-form structure is any of `{Tag, Attributes, Content}`, `{Tag, Content}` or `Tag` where:

- Tag = atom()
- Attributes = [{Name, Value}| #xmlAttribute{}]
- Name = atom()
- Value = IOString | atom() | integer()

See also reference manual for xmerl

If you want to add the information about a black Harley Davidsson 1200 cc Sportster motorcycle from 2003 that is in shape as new in the motorcycles.xml document you can put the data in a simple-form data structure like:

```
Data =
  {bike,
     [{year,"2003"},{color,"black"},{condition,"new"}],
     [{name,
         [{manufacturer,["Harley Davidsson"]},
          {brandName,["XL1200C"]},
          {additionalName,["Sportster"]}]},
      {engine,
         ["V-engine, 2-cylinders, 1200 cc"]},
      {kind,["custom"]},
      {drive,["belt"]}]}
```

In order to append this data to the end of the motorcycles.xml document you have to parse the file and add Data to the end of the root element content.

```
    {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&amp;amp;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>X
```

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&amp;amp;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>
```

The generated XML above was formatted for readability. Another exporter which indents the code with 2 spaces can also be used. In order to use it one only needs to change the export-module:

```
    ...
      Export=xmerl:export_simple([NewRootEl],xmerl_xml_indent,[{prolog,Prolog}]),
    ...
```

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:

```
2> {Doc,Misc}=xmerl_scan:file('motorcycles.xml').
{{xmlElement,motorcycles,
             motorcycles,
             [],
             {xmlNamespace,[],[]},
             [],
             1,
             [],
             [{xmlText,[{motorcycles,1}],1,[],"\
  ",text},
              {xmlElement,bike,
...
3> DocHtml=xmerl:export([Doc],xmerl_html).
["<!DOCTYPE HTML PUBLIC \\"",
 "-//W3C//DTD HTML 4.01 Transitional//EN",
 "\\"",
 [],
 ">\
",
 [["<","motorcycles",">"],
   ["\
  ",
    ["<",
      "bike",
      [[" ","year","=\\"","2000","\\""],[" ","color","=\\"","black","\\""]],
      ">"],
...
```

Will give the result **result_export.html**

Perhaps you want to do something more arranged for human reading. Suppose that you want to list all different brands in the beginning with links to each group of motorcycles. You also want all motorcycles sorted by brand, then some flashy colors on top of it. Thus you rearrange the order of the elements and put in arbitrary HTML tags. This is possible to do by means of the **XSL Transformation (XSLT)** like functionality in xmerl.

Even though the following example shows one way to transform data from XML to HTML it also applies to transformations to other formats.

`xmerl_xs` does not implement the entire XSLT specification but the basic functionality. For all details see the reference manual

First, some words about the xmerl_xs functionality:

You need to write template functions to be able to control what kind of output you want. Thus if you want to encapsulate a `bike` element in <p> tags you simply write a function:

```
template(E = #xmlElement{name='bike'}) ->
    ["<p>",xslapply(fun template/1,E),"</p>"];
```

With `xslapply` you tell the XSLT processor in which order it should traverse the XML structure. By default it goes in preorder traversal, but with the following we make a deliberate choice to break that order:

```
template(E = #xmlElement{name='bike'}) ->
    ["<p>",xslapply(fun template/1,select("bike/name/manufacturer")),"</p>"];
```

If you want to output the content of an XML element or an attribute you will get the value as a string by the `value_of` function:

```
template(E = #xmlElement{name='motorcycles'}) ->
    ["<p>",value_of(select("bike/name/manufacturer",E),"</p>"];
```

In the xmerl_xs functions you can provide a select(String) call, which is an **XPath** functionality. For more details see the xmerl_xs **tutorial**.

Now, back to the example where we wanted to make the output more arranged. With the template:

```
template(E = #xmlElement{name='motorcycles'}) ->
    [    "<head>\
<title>motorcycles</title>\
</head>\
",
         "<body>\
",
\011 "<h1>Used Motorcycles</h1>\
",
\011 "<ul>\
",
\011 remove_duplicates(value_of(select("bike/name/manufacturer",E))),
\011 "\
</ul>\
",
\011 sort_by_manufacturer(xslapply(fun template/1, E)),
         "</body>\
",
\011 "</html>\
"];
```

We match on the top element and embed the inner parts in an HTML body. Then we extract the string values of all motorcycle brands, sort them and removes duplicates by `remove_duplicates(value_of(select("bike/name/manufacturer", E)))`. We also process the substructure of the top element and pass it to a function that sorts all motorcycle information by brand according to the task formulation in the beginning of this example.

The next template matches on the `bike` element:

```
template(E = #xmlElement{name='bike'}) ->
    {value_of(select("name/manufacturer",E)),["<dt>",xslapply(fun template/1,select("name",E)),"</dt>",
    "<dd><ul>\
",
    "<li style="color:green">Manufacturing year: ",xslapply(fun template/1,select("@year",E)),"</li>\
",
    "<li style="color:red">Color: ",xslapply(fun template/1,select("@color",E)),"</li>\
",
    "<li style="color:blue">Shape : ",xslapply(fun template/1,select("@condition",E)),"</li>\
",
    "</ul></dd>\
"]};
```

This creates a tuple with the brand of the motorcycle and the output format. We use the brand name only for sorting purpose. We have to end the template function with the "built in clause" `template(E) -> built_in_rules(fun template/1, E).`

The entire program is motorcycles2html.erl:

```
%%%----------------------------------------------------------------------
%%% File    : motorcycles2html.erl
%%% Author  : Bertil Karlsson <bertil@localhost.localdomain>
%%% Description :
%%%
%%% Created :  2 Sep 2004 by Bertil Karlsson <bertil@localhost.localdomain>
%%%----------------------------------------------------------------------
-module(motorcycles2html).

-include_lib("xmerl/include/xmerl.hrl").

-import(xmerl_xs,
 [ xslapply/2, value_of/1, select/2, built_in_rules/2 ]).

-export([process_xml/1,process_to_file/2,process_to_file/1]).

process_xml(Doc) ->
    template(Doc).

process_to_file(FileName) ->
    process_to_file(FileName,'motorcycles.xml').

process_to_file(FileName,XMLDoc) ->
    case file:open(FileName,[write]) of
 {ok,IOF} ->
     {XMLContent,_} = xmerl_scan:file(XMLDoc),
     TransformedXML=process_xml(XMLContent),
     io:format(IOF,"~s",[TransformedXML]),
     file:close(IOF);
 {error,Reason} ->
     io:format("could not open file due to ~p.~n",[Reason])
    end.

%%% templates
template(E = #xmlElement{name='motorcycles'}) ->
    [    "<head>\n<title>motorcycles</title>\n</head>\n",
         "<body>\n",
  "<h1>Used Motorcycles</h1>\n",
  "<ul>\n",
  remove_duplicates(value_of(select("bike/name/manufacturer",E))),
  "\n</ul>\n",
  sort_by_manufacturer(xslapply(fun template/1, E)),
         "</body>\n",
  "</html>\n"];
template(E = #xmlElement{name='bike'}) ->
    {value_of(select("name/manufacturer",E)),["<dt>",xslapply(fun template/1,select("name",E)),"</dt>",
    "<dd><ul>\n",
    "<li style=\"color:green\">Manufacturing year: ",xslapply(fun template/1,select("@year",E)),"</li>\n",
    "<li style=\"color:red\">Color: ",xslapply(fun template/1,select("@color",E)),"</li>\n",
    "<li style=\"color:blue\">Shape : ",xslapply(fun template/1,select("@condition",E)),"</li>\n",
    "</ul></dd>\n"]};
template(E) -> built_in_rules(fun template/1, E).


%%%%%%%%%% helper routines

%% sorts on the bike name element, unwraps the bike information and
%% inserts a line feed and indentation on each bike element.
sort_by_manufacturer(L) ->
    Tuples=[X1||X1={_,_} <- L],
    SortedTS = lists:keysort(1,Tuples),
    InsertRefName_UnWrap=
 fun([{[Name],V}|Rest],Name,F)->
  [V|F(Rest,Name,F)];
    ([{[Name],V}|Rest],_PreviousName,F) ->
```

```
  [["<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.

xmerl_scan

Erlang module

This module is the interface to the XML parser, it handles XML 1.0. The XML parser is activated through `xmerl_scan:string/[1,2]` or `xmerl_scan:file/[1,2]`. It returns records of the type defined in xmerl.hrl. See also **tutorial** on customization functions.

DATA TYPES

document() = xmlElement() | xmlDocument()

global_state() = #xmerl_scanner{}

option_list() = [{atom(), term()}]

xmlDocument() = #xmlDocument{}

xmlElement() = #xmlElement{}

Exports

```
accumulate_whitespace(T::string(), S::global_state(), X3::atom(),
Acc::string()) -> {Acc, T1, S1}
```

Function to accumulate and normalize whitespace.

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

Equivalent to cont_state(ContinuationState, S).

```
cont_state(X::ContinuationState, S::global_state()) -> global_state()
```

For controlling the ContinuationState, to be used in a continuation function, and called when the parser encounters the end of the byte stream. See **tutorial** on customization functions.

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

Equivalent to event_state(EventState, S).

```
event_state(X::EventState, S::global_state()) -> global_state()
```

For controlling the EventState, to be used in an event function, and called at the beginning and at the end of a parsed entity. See **tutorial** on customization functions.

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

Equivalent to fetch_state(FetchState, S).

```
fetch_state(X::FetchState, S::global_state()) -> global_state()
```

For controlling the FetchState, to be used in a fetch function, and called when the parser fetch an external resource (eg. a DTD). See **tutorial** on customization functions.

```
file(Filename::string()) -> {xmlElement(), Rest}
```

Types:

`Rest = list()`

Equivalent to file(Filename, []).

```
file(Filename::string(), Options::option_list()) -> {document(),
Rest::list()} | {error, Reason::term()}
```

Parse file containing an XML document

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

Equivalent to hook_state(HookState, S).

```
hook_state(X::HookState, S::global_state()) -> global_state()
```

For controlling the HookState, to be used in a hook function, and called when the parser has parsed a complete entity. See **tutorial** on customization functions.

```
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(UserState::term(), S::global_state()) -> global_state()
```

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

xmerl

Erlang module

Functions for exporting XML data to an external format.

Exports

`callbacks(Module) -> Result`

Types:

`Module = atom()`
`Result = [atom()]`

Find the list of inherited callback modules for a given module.

`export(Content, Callback) -> ExportedFormat`

Equivalent to export(Data, Callback, []).

`export(Content, Callback, RootAttributes) -> ExportedFormat`

Types:

`Content = [Element]`
`Callback = atom()`
`RootAttributes = [XmlAttributes]`

Exports normal, well-formed XML content, using the specified callback-module.

`Element` is any of:

- `#xmlText{}`
- `#xmlElement{}`
- `#xmlPI{}`
- `#xmlComment{}`
- `#xmlDecl{}`

(See `xmerl.hrl` for the record definitions.) Text in `#xmlText{}` elements can be deep lists of characters and/or binaries.

`RootAttributes` is a list of `#xmlAttribute{}` attributes for the `#root#` element, which implicitly becomes the parent of the given `Content`. The tag-handler function for `#root#` is thus called with the complete exported data of `Content`. Root attributes can be used to specify e.g. encoding or other metadata of an XML or HTML document.

The `Callback` module should contain hook functions for all tags present in the data structure. A hook function must have the following format:

```
Tag(Data, Attributes, Parents, E)
```

where `E` is the corresponding `#xmlElement{}`, `Data` is the already-exported contents of `E` and `Attributes` is the list of `#xmlAttribute{}` records of `E`. Finally, `Parents` is the list of parent nodes of `E`, on the form `[{ParentTag::atom(), ParentPosition::integer()}]`.

The hook function should return either the data to be exported, or a tuple `{'#xml-alias#', NewTag::atom()}`, or a tuple `{'#xml-redefine#', Content}`, where `Content` is a content list (which can be on simple-form; see `export_simple/2` for details).

A callback module can inherit definitions from other callback modules, through the required function `'#xml-interitance#() -> [ModuleName::atom()]`.

See also: export/2, export_simple/3.

`export_content(Es::Content, Callbacks) -> term()`

Types:

`Content = [Element]`
`Callback = [atom()]`

Exports normal XML content directly, without further context.

`export_element(E, CB) -> term()`

Exports a normal XML element directly, without further context.

`export_element(E, CallbackModule, CallbackState) -> ExportedFormat`

For on-the-fly exporting during parsing (SAX style) of the XML document.

`export_simple(Content, Callback) -> ExportedFormat`

Equivalent to export_simple(Content, Callback, []).

`export_simple(Content, Callback, RootAttrs::RootAttributes) -> ExportedFormat`

Types:

`Content = [Element]`
`Callback = atom()`
`RootAttributes = [XmlAttributes]`

Exports "simple-form" XML content, using the specified callback-module.

`Element` is any of:

- `{Tag, Attributes, Content}`
- `{Tag, Content}`
- `Tag`
- `IOString`
- `#xmlText{}`
- `#xmlElement{}`
- `#xmlPI{}`
- `#xmlComment{}`
- `#xmlDecl{}`

where

- `Tag = atom()`
- `Attributes = [{Name, Value}]`
- `Name = atom()`
- `Value = IOString | atom() | integer()`

Normal-form XML elements can thus be included in the simple-form representation. Note that content lists must be flat. An `IOString` is a (possibly deep) list of characters and/or binaries.

`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.

xmerl_xs

Erlang module

Erlang has similarities to XSLT since both languages have a functional programming approach. Using `xmerl_xpath` it is possible to write XSLT like transforms in Erlang.

XSLT stylesheets are often used when transforming XML documents, to other XML documents or (X)HTML for presentation. XSLT contains quite many functions and learning them all may take some effort. This document assumes a basic level of understanding of XSLT.

Since XSLT is based on a functional programming approach with pattern matching and recursion it is possible to write similar style sheets in Erlang. At least for basic transforms. This document describes how to use the XPath implementation together with Erlangs pattern matching and a couple of functions to write XSLT like transforms.

This approach is probably easier for an Erlanger but if you need to use real XSLT stylesheets in order to "comply to the standard" there is an adapter available to the Sablotron XSLT package which is written i C++. See also the **Tutorial**.

Exports

`built_in_rules(Fun, E) -> List`

The default fallback behaviour. Template funs should end with:
`template(E) -> built_in_rules(fun template/1, E).`

`select(String::string(), E) -> E`

Extracts the nodes from the xml tree according to XPath.

See also: value_of/1.

`value_of(E) -> List`

Types:

`E = term()`

Concatenates all text nodes within the tree.

Example:

```
<xsl:template match="title">
  <div align="center">
    <h1><xsl:value-of select="." /></h1>
  </div>
</xsl:template>
```

becomes:

```
template(E = #xmlElement{name='title'}) ->
  ["<div align="center"><h1>",
    value_of(select(".", E)), "</h1></div>"]
```

`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>"];
```

xmerl_eventp

Erlang module

Simple event-based front-ends to xmerl_scan for processing of XML documents in streams and for parsing in SAX style. Each contain more elaborate settings of xmerl_scan that makes usage of the customization functions.

DATA TYPES

option_list() = [{atom(), term()}]

xmlElement() = #xmlElement{}

Exports

`file_sax(Fname::string(), CallBackModule::atom(), UserState, Options::option_list()) -> NewUserState`

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.

`stream(Fname::string(), Options::option_list()) -> {xmlElement(), list()} | {error, Reason::term()}`

Parse file containing an XML document as a stream, DOM 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` and `close_fun` options cannot be user defined using this parser.

`stream_sax(Fname, CallBack::CallBackModule, UserState, Options) -> xmlElement()`

Types:

`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.

xmerl_xpath

Erlang module

The xmerl_xpath module handles the entire XPath 1.0 spec. XPath expressions typically occur in XML attributes and are used to address parts of an XML document. The grammar is defined in `xmerl_xpath_parse.yrl`. The core functions are defined in `xmerl_xpath_pred.erl`.

Some useful shell commands for debugging the XPath parser

```
c(xmerl_xpath_scan).
yecc:yecc("xmerl_xpath_parse.yrl", "xmerl_xpath_parse", true, []).
c(xmerl_xpath_parse).

xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("position() > -1")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("5 * 6 div 2")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("5 + 6 mod 2")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("5 * 6")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("-----6")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("parent::node()")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("descendant-or-self::node()")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("parent::processing-instruction('foo')")).
```

DATA TYPES

docEntity() = #xmlDocument{} | [docNodes()]

docNodes() = #xmlElement{} | #xmlAttribute{} | #xmlText{} | #xmlPI{} | #xmlComment{} | #xmlNsNode{}

nodeEntity() = #xmlElement{} | #xmlAttribute{} | #xmlText{} | #xmlPI{} | #xmlComment{} | #xmlNsNode{} | #xmlDocument{}

option_list() = [{atom(), term()}]

parentList() = [{atom(), integer()}]

xPathString() = string()

Exports

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

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

`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

xmerl_xsd

Erlang module

Interface module for XML Schema validation. It handles the W3.org **specifications** of XML Schema second edition 28 october 2004. For an introduction to XML Schema study **part 0.** An XML structure is validated by xmerl_xsd:validate/[2,3]. ----------

DATA TYPES

filename() = string()

global_state() = #xsd_state{}

option_list() = [{xsdbase, filename()} | {atom(), term()}]

Exports

`file2state(FileName) -> {ok, State} | {error, Reason}`

Types:

State = global_state()
FileName = string()

Reads the schema state with all information of the processed schema from a file created with `state2file/[1,2]`. The format of this file is internal. The state can then be used validating an XML document.

`format_error(L::Errors) -> Result`

Types:

Errors = tuple() | [tuple()]
Result = string() | [string()]

Formats error descriptions to human readable strings.

`process_schema(Schema) -> Result`

Equivalent to process_schema(Schema, []).

`process_schema(Schema, Options) -> Result`

Types:

Schema = string()
Result = {ok, State} | {error, Reason}
State = global_state()
Reason = [ErrorReason] | ErrorReason
Options = option_list()

Reads the referenced XML schema and checks that it is valid. Returns the `global_state()` with schema info or an error reason. The error reason may be a list of several errors or a single error encountered during the processing.

`process_schemas(Schemas) -> Result`

Equivalent to process_schema(Schemas, []).

`process_schemas(Schemas, Options) -> Result`

Types:

`Schemas = [{NameSpace, string()} | Schemas] | []`
`Result = {ok, State} | {error, Reason}`
`Reason = [ErrorReason] | ErrorReason`
`Options = option_list()`

Reads the referenced XML schemas and controls they are valid. Returns the `global_state()` with schema info or an error reason. The error reason may be a list of several errors or a single error encountered during the processing.

`process_validate(Schema, Xml::Element) -> Result`

Equivalent to process_validate(Schema, Xml, []).

`process_validate(Schema, Xml::Element, Opts::Options) -> Result`

Types:

`Schema = string()`
`Element = XmlElement`
`Options = option_list()`
`Result = {ValidXmlElement, State} | {error, Reason}`
`Reason = [ErrorReason] | ErrorReason`

Validates a parsed well-formed XML element towards an XML schema.

Validates in two steps. First it processes the schema, saves the type and structure info in an ets table and then validates the element towards the schema.

Usage example:

```
1>{E,_} = xmerl_scan:file("my_XML_document.xml").
2>{E2,_} = xmerl_xsd:process_validate("my_XML_Schema.xsd",E).
```

Observe that E2 may differ from E if for instance there are default values defined in `my_XML_Schema.xsd`.

`state2file(S::State) -> ok | {error, Reason}`

Same as state2file(State,SchemaName)

The name of the saved file is the same as the name of the schema, but with `.xss` extension.

`state2file(S::State, FileName) -> ok | {error, Reason}`

Types:

`State = global_state()`
`FileName = string()`

Saves the schema state with all information of the processed schema in a file. You can provide the file name for the saved state. FileName is saved with the `.xss` extension added.

`validate(Xml::Element, State) -> Result`

Equivalent to validate(Element, State, []).

```
validate(Element, State, Options) -> Result
```

Types:

```
Element = #xmlElement{}
Options = option_list()
Result = {ValidElement, global_state()} | {error, Reasons}
ValidElement = #xmlElement{}
State = global_state()
Reasons = [ErrorReason] | ErrorReason
ErrorReason = term()
```

Validates a parsed well-formed XML element (Element).

A call to validate/2 or validate/3 must provide a well formed parsed XML element `#xmlElement{}` and a State, `global_state()`, which holds necessary information from an already processed schema. Thus validate enables reuse of the schema information and therefore if one shall validate several times towards the same schema it reduces time consumption.

The result, ValidElement, is the valid element that conforms to the post-schema-validation infoset. When the validator finds an error it tries to continue and reports a list of all errors found. In those cases an unexpected error is found it may cause a single error reason.

Usage example:

```
1>{E,_} = xmerl_scan:file("my_XML_document.xml").
2>{ok,S} = xmerl_xsd:process_schema("my_XML_Schema.xsd").
3>{E2,_} = xmerl_xsd:validate(E,S).
```

Observe that E2 may differ from E if for instance there are default values defined in `my_XML_Schema.xsd`.

xmerl_sax_parser

Erlang module

A SAX parser for XML that sends the events through a callback interface. SAX is the **Simple API for XML**, originally a Java-only API. SAX was the first widely adopted API for XML in Java, and is a **de facto** standard where there are versions for several programming language environments other than Java.

DATA TYPES

`option()`

Options used to customize the behaviour of the parser. Possible options are:

`{continuation_fun, ContinuationFun}`
: ContinuationFun is a call back function to decide what to do if the parser runs into EOF before the document is complete.

`{continuation_state, term()}`
: State that is accessible in the continuation call back function.

`{event_fun, EventFun}`
: EventFun is the call back function for parser events.

`{event_state, term()}`
: State that is accessible in the event call back function.

`{file_type, FileType}`
: Flag that tells the parser if it's parsing a DTD or a normal XML file (default normal).

- `FileType = normal | dtd`

`{encoding, Encoding}`
: Set default character set used (default UTF-8). This character set is used only if not explicitly given by the XML document.

- `Encoding = utf8 | {utf16,big} | {utf16,little} | latin1 | list`

`skip_external_dtd`
: Skips the external DTD during parsing. This option is the same as {external_entities, none} and {fail_undeclared_ref, false} but just for the DTD.

`disallow_entities`
: Implies that parsing fails if an ENTITY declaration is found.

`{entity_recurse_limit, N}`
: Sets how many levels of recursion that is allowed for entities. Default is 3 levels.

`{external_entities, AllowedType}`
: Sets which types of external entities that should be allowed, if not allowed it's just skipped.

- `AllowedType = all | file | none`

`{fail_undeclared_ref, Boolean}`
: Decides how the parser should behave when an undeclared reference is found. Can be useful if one has turned of external entities so that an external DTD is not parsed. Default is true.

`{discard_ws_before_xml_document, Boolean}`
: Discard whitespace before `xml` tag instead of returning a fatal error. Default is false.

`event()`

The SAX events that are sent to the user via the callback.

`startDocument`
Receive notification of the beginning of a document. The SAX parser will send this event only once before any other event callbacks.

`endDocument`
Receive notification of the end of a document. The SAX parser will send this event only once, and it will be the last event during the parse.

`{startPrefixMapping, Prefix, Uri}`
Begin the scope of a prefix-URI Namespace mapping. Note that start/endPrefixMapping events are not guaranteed to be properly nested relative to each other: all startPrefixMapping events will occur immediately before the corresponding startElement event, and all endPrefixMapping events will occur immediately after the corresponding endElement event, but their order is not otherwise guaranteed. There will not be start/endPrefixMapping events for the "xml" prefix, since it is predeclared and immutable.

- `Prefix = string()`
- `Uri = 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.

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

Exports

`Module: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.

`Module: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.