xmerl Application

version 1.0

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

Chapter 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[1] site you will find the XML 1.0 specification[2] and other related specs. One site were you can find tutorials on XML and related specs is ZVON.org[3].

However, here you will find some examples of how to use xmerl and some scenarios in which it can be used. A detailed description of the user interface can be found in the reference manual[4].

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 markup names that are encoded Unicode characters not mapping on ASCII.

[1] URL: `http://www.w3.org`
[2] URL: `http://www.w3.org/TR/REC-xml/`
[3] URL: `http://www.zvon.org`
[4] URL: `application_frame.html`

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 convienient 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. Then it is easy to for instance 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 usage examples of these three scenarios 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 markup 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 are 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,[],"\nGrand Danois\n",text}],
            ...
            }
```

Where the content of the top element is: `[{xmlText,[{dog,1}],1,[],"\nGrand Danois\n",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 charactes between markup as `space`, `tab` and `line feed` are normalized and returned as xmlText records.

Errors

An unsucsessful 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

If you want to parse an XML file you run it in the Erlang shell like:

```
3> {ParsResult,Misc}=xmerl_scan:file("people.xml"[5]).
{{xmlElement,people,
             people,
             [],
             {xmlNamespace,[],[]},
             [],
             1,
             [],
             [{xmlText,[{people,1}],1,[],"\n  ",text},
              {xmlElement,person,
                          person,
                          [],
                          {xmlNamespace,[],[]},
                          [{people,1}],
                          2,
                          [{xmlAttribute,born,[],[],[],[]|...},
                           {xmlAttribute,died,[],[],[]|...}],
                          [{xmlText,[{person,2},{people|...}],
                                    1,
                                    []|...},
                           {xmlElement,name,name,[]|...},
                           {xmlText,[{...}|...],3|...},
                           {xmlElement,profession|...},
                           {xmlText|...},
                           {...}|...],
                          [],
                          ".",
                          undeclared},
              {xmlText,[{people,1}],3,[],"\n  ",text},
              {xmlElement,person,
                          person,
                          [],
```

[5] URL: `people.txt`

```
                          {xmlNamespace,[],[]},
                          [{people,1}],
                          4,
                          [{xmlAttribute,born,[],[]|...},
                           {xmlAttribute,died,[]|...}],
                          [{xmlText,[{...}|...],1|...},
                           {xmlElement,name|...},
                           {xmlText|...},
                           {...}|...],
                          [],
                          ".",
                          undeclared},
               {xmlText,[{people,1}],5,[],"\n",text}],
              [],
              ".",
              undeclared},
 "\n"}
4>
```

If you instead receive 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[6].

1.1.4 Example: Extracting Data From XML Content

In this and the following examples we use the XML file ”motorcycles.xml”[7] and the corresponding DTD ”motorcycles.dtd”[8].

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 has 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 exist 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 markup results in an xmlText record.

[6] URL: `xmerl_scan.html`
[7] URL: `motorcycles.txt`
[8] URL: `motorcycles_dtd.txt`

1.1.5 Example: Create XML Out Of Arbitrary Data

For this task there is 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 apropriate 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 xmlElment 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[9]

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"]}]}
```

[9] URL: `xmerl.html#export_simple-3`

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 new_motorcycles.xml[10]. 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 apropriate 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="\n    "},
          {name,[#xmlText{value="\n      "},
                 {manufacturer,["Harley Davidsson"]},
                 #xmlText{value="\n      "},
                 {brandName,["XL1200C"]},
                 #xmlText{value="\n      "},
                 {additionalName,["Sportster"]},
                 #xmlText{value="\n    "}]},
          {engine,["V-engine, 2-cylinders, 1200 cc"]},
          #xmlText{value="\n    "},
          {kind,["custom"]},
          #xmlText{value="\n    "},
          {drive,["belt"]},
          #xmlText{value="\n  "}]},
   #xmlText{value="\n"}],
...
NewContent=Content++lists:flatten([Data]),
NewRootEl=RootEl#xmlElement{content=NewContent},
...
Prolog = ["<?xml version=\"1.0\" encoding=\"utf-8\" ?>
<!DOCTYPE motorcycles SYSTEM \"motorcycles.dtd\">\n"],
Export=xmerl:export_simple([NewRootEl],xmerl_xml,[{prolog,Prolog}]),
...
```

The result will be new_motorcycles2.xml[11].

[10] URL: new_motorcycles.txt
[11] URL: new_motorcycles2.txt

1.1.6 Example: Transforming XML To HTML

Assume that you want to transform the motorcycles.xml[12] 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,[],"\n  ",text},
              {xmlElement,bike,
...
3> DocHtml=xmerl:export([Doc],xmerl_html).
["<!DOCTYPE HTML PUBLIC \"",
 "-//W3C//DTD HTML 4.01 Transitional//EN",
 "\"",
 [],
 ">\n",
 [["<","motorcycles",">"],
   ["\n  ",
    [["<",
      "bike",
      [[" ","year","=\"","2000","\""],[" ","color","=\"","black","\""]],
      ">"],
...
```

Will give the result result_export.html[13]

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 colours 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)[14] 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[15]

First, some words about the xmerl_xs functionality:

You need to wright 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 wright a function:

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

[12] URL: `motorcycles.txt`
[13] URL: `result_export.html`
[14] URL: `http://www.w3.org/Style/XSL/`
[15] URL: `xmerl_xs.html`

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[16] functionality. For more details see the xmerl_xs tutorial[17].

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

```
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"];
```

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>\n",
    "<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>\n",
    "<li style=\"color:blue\">Shape : ",xslapply(fun template/1,select("@condition",E)),"</li>\n",
    "</ul></dd>\n"]};
```

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 fuction with the "built in clause" template(E) -> built_in_rules(fun template/1, E).

The entire program is motorcycles2html.erl[18]. If we run it like this motorcycles2html:process_to_file('result_xs.html', 'motorcycles2.xml'). The input is motorcycles2.xml[19] and you can check the result in the output file result_xs.html[20].

[16] URL: http://www.w3.org/TR/xpath
[17] URL: xmerl_xs_examples.html
[18] URL: motorcycles2html.erl
[19] URL: motorcycles2.txt
[20] URL: result_xs.html

1.2 xmerl Release Notes

This document describes the changes made to the xmerl application.

1.2.1 xmerl 1.0

Improvements and New Features

- The OTP release of xmerl 1.0 is mainly the same as xmerl-0.20 of http://sowap.sourceforge.net/. It is capable of parsing XML 1.0. There have only been minor improvements: Some bugs that caused an unexpected crash when parsing bad XML. Failure report that also tells which file that caused an error.
  Own Id: OTP-5174

Known Problems

- xmerl cannot fetch external entities that are referenced by an URL. Only references to the local file system are working.
  Own Id: OTP-5173
- xmerl cannot handle markup names that are Unicode encoded characters bigger than 256.
  Own Id: OTP-5172

There are also release notes for older versions[21].

[21] URL: `notes_history.html`

xmerl Reference Manual

Short Summaries

- Erlang Module xmerl [page 14] – Functions for exporting XML data to an external format.
- Erlang Module xmerl_eventp [page 17] – Simple event-based front-ends to xmerl_scan for processing of XML documents in streams and for parsing in SAX style.
- Erlang Module xmerl_scan [page 18] – This module is the interface to the XML parser, it handles XML 1.0.
- Erlang Module xmerl_xpath [page 21] – The xmerl_xpath module handles the entire XPath 1.0 spec XPath expressions typically occurs in XML attributes and are used to addres parts of an XML document.
- Erlang Module xmerl_xs [page 23] – Erlang has similarities to XSLT since both languages have a functional programming approach.

xmerl

The following functions are exported:

- `callbacks(M::atom()) -> [atom()]`
  [page 14] Find the list of inherited callback modules for a given module.
- `export(Data::Content, Callback) -> ExportedFormat`
  [page 14] Equivalent to export(Data, Callback, []).
- `export(Data::Content, Callback, RootAttrs::RootAttributes) -> ExportedFormat`
  [page 14] Exports normal, well-formed XML content, using the specified callback-module.
- `export_content(Es::Content, CBs::Callbacks) -> term()`
  [page 15] Exports normal XML content directly, without further context.
- `export_element(E, CB) -> term()`
  [page 15] Exports a normal XML element directly, without further context.
- `export_element(E, CB::CBs, CBstate::UserState) -> ExportedFormat`
  [page 15] For on-the-fly exporting during parsing (SAX style) of the XML document.
- `export_simple(Data::Content, Callback) -> ExportedFormat`
  [page 15] Equivalent to export_simple(Data, Callback, []).

- `export_simple(Data::Content, Callback, RootAttrs::RootAttributes) -> ExportedFormat`
  [page 15] Exports ”simple-form” XML content, using the specified callback-module.
- `export_simple_content(Data, Callback) -> term()`
  [page 16] Exports simple XML content directly, without further context.
- `export_simple_element(Data, Callback) -> term()`
  [page 16] Exports a simple XML element directly, without further context.

xmerl_eventp

The following functions are exported:

- `file_sax(Fname::string(), CallBackModule::atom(), UserState, Options::option_list()) -> NewUserState`
  [page 17] Parse file containing an XML document, SAX style.
- `stream(Fname::string(), Options::option_list()) -> xmlElement()`
  [page 17] Parse file containing an XML document as a stream, DOM style.
- `stream_sax(Fname, CallBack::CallBackModule, UserState, Options) -> xmlElement()`
  [page 17] Parse file containing an XML document as a stream, SAX style.
- `string_sax(String::list(), CallBackModule::atom(), UserState, Options::option_list()) -> xmlElement()`
  [page 17] Parse file containing an XML document, SAX style.

xmerl_scan

The following functions are exported:

- `accumulate_whitespace(T::string(), S::global_state(), X3::atom(), Acc::string()) -> {Acc, T1, S1}`
  [page 19] Function to accumulate and normalize whitespace.
- `cont_state(S::global_state()) -> global_state()`
  [page 19] Equivalent to cont_state(ContinuationState, S).
- `cont_state(X::ContinuationState, S::global_state()) -> global_state()`
  [page 19] For controlling the ContinuationState, to be used in a continuation function, and called when the parser encounters the end of the byte stream.
- `event_state(S::global_state()) -> global_state()`
  [page 19] Equivalent to event_state(EventState, S).
- `event_state(X::EventState, S::global_state()) -> global_state()`
  [page 19] For controlling the EventState, to be used in an event function, and called at the beginning and at the end of a parsed entity.
- `fetch_state(S::global_state()) -> global_state()`
  [page 19] Equivalent to fetch_state(FetchState, S).
- `fetch_state(X::FetchState, S::global_state()) -> global_state()`
  [page 19] For controlling the FetchState, to be used in a fetch function, and called when the parser fetch an external resource (eg.

- `file(Filename::string()) -> {xmlElement(), Rest}`
  [page 19] Equivalent to file(Filename, []).
- `file(Filename::string(), Options::option_list()) -> {xmlElement(), Rest}`
  [page 19] Parse file containing an XML document.
- `hook_state(S::global_state()) -> global_state()`
  [page 20] Equivalent to hook_state(HookState, S).
- `hook_state(X::HookState, S::global_state()) -> global_state()`
  [page 20] For controlling the HookState, to be used in a hook function, and called when the parser has parsed a complete entity.
- `rules_state(S::global_state()) -> global_state()`
  [page 20] Equivalent to rules_state(RulesState, S).
- `rules_state(X::RulesState, S::global_state()) -> global_state()`
  [page 20] For controlling the RulesState, to be used in a rules function, and called when the parser store scanner information in a rules database.
- `string(Text::list()) -> {xmlElement(), Rest}`
  [page 20] Equivalent to string(Test, []).
- `string(Text::list(), Options::option_list()) -> {xmlElement(), Rest}`
  [page 20] Parse string containing an XML document.
- `user_state(S::global_state()) -> global_state()`
  [page 20] Equivalent to user_state(UserState, S).
- `user_state(X::UserState, S::global_state()) -> global_state()`
  [page 20] For controlling the UserState, to be used in a user function.

xmerl_xpath

The following functions are exported:

- `string(Str, Doc) -> docEntity()`
  [page 21] Equivalent to string(Str, Doc, []).
- `string(Str, Doc, Options) -> docEntity()`
  [page 21] Equivalent to string(Str, Doc, [], Doc, Options).
- `string(Str, Node, Parents, Doc, Options) -> docEntity()`
  [page 22] Extracts the nodes from the parsed XML tree according to XPath.

xmerl_xs

The following functions are exported:

- `built_in_rules(Fun, E) -> List`
  [page 23] The default fallback behaviour.
- `select(String::string(), E) -> E`
  [page 23] Extracts the nodes from the xml tree according to XPath.
- `value_of(E) -> List`
  [page 23] Concatenates all text nodes within the tree.
- `xslapply(Fun::Function, EList::list()) -> List`
  [page 24] xslapply is a wrapper to make things look similar to xsl:apply-templates.

xmerl

Erlang Module

Functions for exporting XML data to an external format.

Exports

`callbacks(M::atom()) -> [atom()]`

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

`export(Data::Content, Callback) -> ExportedFormat`

Equivalent to export(Data, Callback, []) [page 14].

`export(Data::Content, Callback, RootAttrs::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 [page 14], export_simple/3 [page 15].

`export_content(Es::Content, CBs::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, CB::CBs, CBstate::UserState) -> ExportedFormat`

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

`export_simple(Data::Content, Callback) -> ExportedFormat`

Equivalent to export_simple(Data, Callback, []) [page 15].

`export_simple(Data::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 [page 14], export_simple/2 [page 15].

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

Exports simple XML content directly, without further context.

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

Exports a simple XML element directly, without further context.

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.

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()`

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_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 [page ??] on customization functions.

Data Types

abstract datatype: `global_state()` The global state of the scanner, represented by the #xmerl_scanner{} record.

abstract datatype: `option_list()` Options allow to customize the behaviour of the scanner. See also tutorial [page ??] on customization functions.

Possible options are:

{`acc_fun, Fun`} Call back function to accumulate contents of entity.

{`continuation_fun, Fun`} | {`continuation_fun, Fun, ContinuationState`} Call back function to decide what to do if the scanner runs into EOF before the document is complete.

{`event_fun, Fun`} | {`event_fun, Fun, EventState`} Call back function to handle scanner events.

{`fetch_fun, Fun`} | {`fetch_fun, Fun, FetchState`} Call back function to fetch an external resource.

{`hook_fun, Fun`} | {`hook_fun, Fun, HookState`} Call back function to process the document entities once identified.

{`close_fun, Fun`} Called when document has been completely parsed.

{`rules, ReadFun, WriteFun, RulesState`} | {`rules, Rules`} Handles storing of scanner information when parsing.

{`user_state, UserState`} Global state variable accessible from all customization functions

{`fetch_path, PathList`} PathList is a list of directories to search when fetching files. If the file in question is not in the fetch_path, the URI will be used as a file name.

{`space, Flag`} 'preserve' (default) to preserve spaces, 'normalize' to accumulate consecutive whitespace and replace it with one space.

{`line, Line`} To specify starting line for scanning in document which contains fragments of XML.

{`namespace_conformant, Flag`} Controls whether to behave as a namespace conformant XML parser, 'false' (default) to not otherwise 'true'.

{`validation, Flag`} Controls whether to process as a validating XML parser, 'false' (default) to not otherwise 'true'.

{`quiet, Flag`} Set to 'true' if xmerl should behave quietly and not output any information to standard output (default 'false').

{doctype_DTD, DTD} Allows to specify DTD name when it isn't available in the XML document.

{xmlbase, Dir} XML Base directory. If using string/1 default is current directory. If using file/1 default is directory of given file.

{encoding, Enc} Set default character set used (default utf-8). This character set is used only if not explicitly given by the XML declaration. Note: If ucs is not available this MUST be set to a character set that is a true subset of Unicode and where each character only require a single byte. E.g., iso-8859-1 or equivalent

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) [page 19].

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 [page ??] on customization functions.

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

Equivalent to event_state(EventState, S) [page 19].

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 [page ??] on customization functions.

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

Equivalent to fetch_state(FetchState, S) [page 19].

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 [page ??] on customization functions.

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

Types:

- Rest = list()

Equivalent to file(Filename, []) [page 20].

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

Types:

- Rest = list()

Parse file containing an XML document

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

Equivalent to hook_state(HookState, S) [page 20].

`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 [page ??] on customization functions.

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

Equivalent to rules_state(RulesState, S) [page 20].

`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 [page ??] on customization functions.

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

Types:

- Rest = list()

Equivalent to string(Test, []) [page 20].

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

Types:

- Rest = list()

Parse string containing an XML document

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

Equivalent to user_state(UserState, S) [page 20].

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

For controlling the UserState, to be used in a user function. See tutorial [page ??] on customization functions.

xmerl_xpath

Erlang Module

The xmerl_xpath module handles the entire XPath 1.0 spec XPath expressions typically occurs in XML attributes and are used to addres 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() = xmlElement() | xmlAttribute() | xmlText() | xmlPI() | xmlComment()

nodeEntity() = xmlElement() | xmlAttribute() | xmlText() | xmlPI() | xmlNamespace() |
```

abstract datatype: `option_list()` Options allows to customize the behaviour of the XPath scanner.

Possible options are:

{`namespace, #xmlNamespace`} Set namespace nodes, from XmlNamspace, in xmlContext

{`namespace, Nodes`} Set namespace nodes in xmlContext.

Exports

`string(Str, Doc) -> docEntity()`

Equivalent to string(Str, Doc, []) [page 22].

`string(Str, Doc, Options) -> docEntity()`

Equivalent to string(Str, Doc, [], Doc, Options) [page 22].

`string(Str, Node, Parents, Doc, Options) -> docEntity()`

Types:

- Str = xPathString()
- Node = nodeEntity()
- Parents = parentList()
- Doc = nodeEntity()
- Options = option_list()

Extracts the nodes from the parsed XML tree according to XPath.

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. There are a number of brick-sized books written on the topic. XSLT contains quite many functions and learning them all may take some effort, which could be a reason why the author only has reached a basic level of understanding. 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 [page ??].

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 [page 23].

`value_of(E) -> List`

Types:

- E = unknown()

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>"];
```

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.