SWI-Prolog SGML/XML parser

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: J.Wielemaker@uva.nl

February 29, 2012

Abstract

Markup languages are an increasingly important method for data-representation and exchange. This article documents the package sgml, a foreign library for SWI-Prolog to parse SGML and XML documents, returning information on both the document and the document's DTD. The parser is designed to be small, fast and flexible.

Contents

1 Introduction

Markup languages have recently regained popularity for two reasons. One is document exchange, which is largely based on HTML, an instance of SGML, and the other is for data exchange between programs, which is often based on XML, which can be considered a simplified and rationalised version of SGML.

James Clark's SP parser is a flexible SGML and XML parser. Unfortunately it has some drawbacks. It is very big, not very fast, cannot work under event-driven input and is generally hard to program beyond the scope of the well designed generic interface. The generic interface however does not provide access to the DTD, does not allow for flexible handling of input or parsing the DTD independently of a document instance.

The parser described in this document is small (less than 100 kBytes executable on a Pentium), fast (between 2 and 5 times faster than SP), provides access to the DTD, and provides flexible input handling.

The document output is equal to the output produced by *xml2pl*, an SP interface to SWI-Prolog written by Anjo Anjewierden.

2 Bluffer's Guide

This package allows you to parse SGML, XML and HTML data into a Prolog data structure. The high-level interface defined in `sgml` provides access at the file-level, while the low-level interface defined in the foreign module works with Prolog streams. Please use the source of `sgml.pl` as a starting point for dealing with data from other sources than files, such as SWI-Prolog resources, network-sockets, character strings, *etc.* The first example below loads an HTML file.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">

<html>
<head>
<title>Demo</title>
</head>
<body>

<h1 align=center>This is a demo</title>

Paragraphs in HTML need not be closed.

This is called 'omitted-tag' handling.
</body>
</html>
```

```
?- load_html_file('test.html', Term),
   pretty_print(Term).

[ element(html,
          [],
```

```
            [ element(head,
                      [],
                      [ element(title,
                                [],
                                [ 'Demo'
                                ])
                      ]),
              element(body,
                      [],
                      [ '\n',
                        element(h1,
                                [ align = center
                                ],
                                [ 'This is a demo'
                                ]),
                        '\n\n',
                        element(p,
                                [],
                                [ 'Paragraphs in HTML need not be closed.\n'
                                ]),
                        element(p,
                                [],
                                [ 'This is called `omitted-tag\' handling.'
                                ])
                      ])
            ])
].
```

The document is represented as a list, each element being an atom to represent `CDATA` or a term `element`(*Name, Attributes, Content*). Entities (e.g. `<`) are expanded and included in the atom representing the element content or attribute value.[1]

2.1 'Goodies' Predicates

These predicates are for basic use of the library, converting entire and self-contained files in SGML, HTML, or XML into a structured term. They are based on `load_structure/3`.

load_sgml_file(+*Source, -ListOfContent*)
Same as `load_structure`(*Source, ListOfContent, [dialect(sgml)]*).

load_xml_file(+*Source, -ListOfContent*)
Same as `load_structure(Source, ListOfContent, [dialect(xml)]`(.)

load_html_file(+*Source, -Content*)
Load *Source* and parse as HTML. *Source* is either the name of a file or term `stream`(*Handle*).

[1] Up to SWI-Prolog 5.4.x, Prolog could not represent *wide* characters and entities that did not fit in the Prolog characters set were emitted as a term `number`(+*Code*). With the introduction of wide characters in the 5.5 branch this is no longer needed.

Implemented as below. Note that load_html_file/2 re-uses a cached DTD object as defined by dtd/2. As DTD objects may be corrupted while loading errornous documents sharing is undesirable if the documents are not known to be correct. See dtd/2 for details.

```
load_html_file(Source, Term) :-
        dtd(html, DTD),
        load_structure(Source, Term,
                       [ dtd(DTD),
                         dialect(sgml),
                         shorttag(false)
                       ]).
```

3 Predicate Reference

3.1 Loading Structured Documents

SGML or XML files are loaded through the common predicate load_structure/3. This is a predicate with many options. For simplicity a number of commonly used shorthands are provided: load_sgml_file/2, load_xml_file/2, and load_html_file/2.

load_structure(+*Source, -ListOfContent, +Options*)

Parse *Source* and return the resulting structure in *ListOfContent*. *Source* is either a term of the format stream(*StreamHandle*) or a file-name. *Options* is a list of options controlling the conversion process.

A proper XML document contains only a single toplevel element whose name matches the document type. Nevertheless, a list is returned for consistency with the representation of element content. The *ListOfContent* consists of the following types:

Atom

Atoms are used to represent CDATA. Note this is possible in SWI-Prolog, as there is no length-limit on atoms and atom garbage collection is provided.

element(*Name, ListAttributes, ListOfContent*)

Name is the name of the element. Using SGML, which is case-insensitive, all element names are returned as lowercase atoms.

ListOfAttributes is a list of *Name*=*Value* pairs for attributes. Attributes of type CDATA are returned literal. Multi-valued attributes (NAMES, *etc.*) are returned as a list of atoms. Handling attributes of the types NUMBER and NUMBERS depends on the setting of the number(+*NumberMode*) attribute through set_sgml_parser/2 or load_structure/3. By default they are returned as atoms, but automatic conversion to Prolog integers is supported. *ListOfContent* defines the content for the element.

sdata(*Text*)

If an entity with declared content-type SDATA is encountered, this term is returned holding the data in *Text*.

ndata(*Text*)

If an entity with declared content-type NDATA is encountered, this term is returned holding the data in *Text*.

pi(*Text*)
: If a processing instruction is encountered (`<?...?>`), *Text* holds the text of the processing instruction. Please note that the `<?xml ...?>` instruction is handled internally.

The *Options* list controls the conversion process. Currently defined options are:

dtd(*?DTD*)
: Reference to a DTD object. If specified, the `<!DOCTYPE ...>` declaration is ignored and the document is parsed and validated against the provided DTD. If provided as a variable, the created DTD is returned. See section 3.5.

dialect(+*Dialect*)
: Specify the parsing dialect. Supported are `sgml` (default), `xml` and `xmlns`. See section 3.3 for details on the differences.

shorttag(+*Bool*)
: Define whether SHORTTAG abbreviation is accepted. The default is true for SGML mode and false for the XML modes. Without SHORTTAG, a `/` is accepted with warning as part of an unquoted attribute-value, though `/>` still closes the element-tag in XML mode. It may be set to false for parsing HTML documents to allow for unquoted URLs containing `/`.

space(+*SpaceMode*)
: Sets the 'space-handling-mode' for the initial environment. This mode is inherited by the other environments, which can override the inherited value using the XML reserved attribute `xml:space`. See section 3.2.

number(+*NumberMode*)
: Determines how attributes of type `NUMBER` and `NUMBERS` are handled. If `token` (default) they are passed as an atom. If `integer` the parser attempts to convert the value to an integer. If successful, the attribute is passed as a Prolog integer. Otherwise it is still passed as an atom. Note that SGML defines a numeric attribute to be a sequence of digits. The `-` sign is not allowed and `1` is different from `01`. For this reason the default is to handle numeric attributes as tokens. If conversion to integer is enabled, negative values are silently accepted.

defaults(+*Bool*)
: Determines how default and fixed values from the DTD are used. By default, defaults are included in the output if they do not appear in the source. If `false`, only the attributes occurring in the source are emitted.

entity(+*Name, +Value*)
: Defines (overwrites) an entity definition. At the moment, only `CDATA` entities can be specified with this construct. Multiple entity options are allowed.

file(+*Name*)
: Sets the name of the file on which errors are reported. Sets the linenumber to 1.

line(+*Line*)
: Sets the starting line-number for reporting errors.

max_errors(+*Max*)
: Sets the maximum number of errors. If this number is reached, an exception of the format below is raised. The default is 50. Using `max_errors`(*-1*) makes the parser continue, no matter how many errors it encounters.

`error`(*limit_exceeded(max_errors, Max), _*)

3.2 Handling white-space

SGML2PL has four modes for handling white-space. The initial mode can be switched using the `space`(*SpaceMode*) option to `load_structure/3` and `set_sgml_parser/2`. In XML mode, the mode is further controlled by the `xml:space` attribute, which may be specified both in the DTD and in the document. The defined modes are:

space(*sgml*)
: In SGML, newlines at the start and end of an element are removed.[2] This is the default mode for the SGML dialect.

space(*preserve*)
: White space is passed literally to the application. This mode leaves all white space handling to the application. This is the default mode for the XML dialect.

space(*default*)
: In addition to `sgml` space-mode, all consequtive white-space is reduced to a single space-character. This mode canonises all white space.

space(*remove*)
: In addition to `default`, all leading and trailing white-space is removed from `CDATA` objects. If, as a result, the `CDATA` becomes empty, nothing is passed to the application. This mode is especially handy for processing 'data-oriented' documents, such as RDF. It is not suitable for normal text documents. Consider the HTML fragment below. When processed in this mode, the spaces between the three modified words are lost. This mode is not part of any standard; XML 1.0 allows only `default` and `preserve`.

```
Consider adjacent <b>bold</b> <ul>and</ul> <it>italic</it> words.
```

3.3 XML documents

The parser can operate in two modes: `sgml` mode and `xml` mode, as defined by the `dialect`(*Dialect*) option. Regardless of this option, if the first line of the document reads as below, the parser is switched automatically into XML mode.

```
<?xml ... ?>
```

Currently switching to XML mode implies:

- *XML empty elements*
  The construct `<element [attribute...] />` is recognised as an empty element.
- *Predefined entities*
  The following entitities are predefined: `lt` (<), `gt` (>), `amp` (&), `apos` (') and `quot` (").

[2] In addition, newlines at the end of lines containing only markup should be deleted. This is not yet implemented.

- *Case sensitivity*
  In XML mode, names are treated case-sensitive, except for the DTD reserved names (i.e. `ELEMENT`, *etc.*).

- *Character classes*
  In XML mode, underscores (_) and colon (`:`) are allowed in names.

- *White-space handling*
  White space mode is set to `preserve`. In addition to setting white-space handling at the toplevel the XML reserved attribute `xml:space` is honoured. It may appear both in the document and the DTD. The `remove` extension is honoured as `xml:space` value. For example, the DTD statement below ensures that the `pre` element preserves space, regardless of the default processing mode.

```
<!ATTLIST pre xml:space nmtoken #fixed preserve>
```

3.3.1 XML Namespaces

Using the *dialect* `xmlns`, the parser will interpret XML namespaces. In this case, the names of elements are returned as a term of the format

URL:*LocalName*

If an identifier has no namespace and there is no default namespace it is returned as a simple atom. If an identifier has a namespace but this namespace is undeclared, the namespace name rather than the related URL is returned.

Attributes declaring namespaces (`xmlns:`⟨*ns*⟩=⟨*url*⟩) are reported as if `xmlns` were not a defined resource.

In many cases, getting attribute-names as *url*:*name* is not desirable. Such terms are hard to unify and sometimes multiple URLs may be mapped to the same identifier. This may happen due to poor version management, poor standardisation or because the the application doesn't care too much about versions. This package defines two call-backs that can be set using `set_sgml_parser/2` to deal with this problem.

The call-back `xmlns` is called as XML namespaces are noticed. It can be used to extend a canonical mapping for later use by the `urlns` call-back. The following illustrates this behaviour. Any namespace containing `rdf-syntax` in its URL or that is used as `rdf` namespace is canonised to `rdf`. This implies that any attribute and element name from the RDF namespace appears as `rdf:\bnfmeta{name}`

```
:- dynamic
        xmlns/3.

on_xmlns(rdf, URL, _Parser) :- !,
        asserta(xmlns(URL, rdf, _)).
on_xmlns(_, URL, _Parser) :-
        sub_atom(URL, _, _, _, 'rdf-syntax'), !,
        asserta(xmlns(URL, rdf, _)).
```

```
load_rdf_xml(File, Term) :-
        load_structure(File, Term,
                       [ dialect(xmlns),
                         call(xmlns, on_xmlns),
                         call(urlns, xmlns)
                       ]).
```

The library provides `iri_xml_namespace/3` to break down an IRI into its namespace and localname:

iri_xml_namespace(+*IRI, -Namespace, -Localname*) *[det]*

Split an IRI (Unicode URI) into its *Namespace* (an IRI) and *Localname* (a Unicode XML name, see `xml_name/2`). The *Localname* is defined as the longest last part of the IRI that satisfies the syntax of an XML name. With IRI schemas that are designed to work with XML namespaces, this will typically break the IRI on the last `#` or `/`. Note however that this can produce unexpected results. E.g., in the example below, one might expect the namespace to be `http://example.com/images#`, but an XML name cannot start with a digit.

```
?- iri_xml_namespace('http://example.com/images#12345', NS, L).
NS = 'http://example.com/images#12345',
L = ''.
```

As we see from the example above, the *Localname* can be the empty atom. Similarly, *Namespace* can be the empty atom if *IRI* is an XML name. Applications will often have to check for either or both these conditions. We decided against failing in these conditions because the application typically wants to know which of the two conditions (empty namespace or empty localname) holds. This predicate is often used for generating RDF/XML from an RDF graph.

iri_xml_namespace(+*IRI, -Namespace*) *[det]*

Same as `iri_xml_namespace/3`, but avoids creating an atom for the *Localname*.

3.4 DTD-Handling

The DTD (**D**ocument **T**ype **D**efinition) is a separate entity in sgml2pl, that can be created, freed, defined and inspected. Like the parser itself, it is filled by opening it as a Prolog output stream and sending data to it. This section summarises the predicates for handling the DTD.

new_dtd(+*DocType, -DTD*)

Creates an empty DTD for the named *DocType*. The returned DTD-reference is an opaque term that can be used in the other predicates of this package.

free_dtd(+*DTD*)

Deallocate all resources associated to the DTD. Further use of *DTD* is invalid.

load_dtd(+*DTD, +File*)

Define the DTD by loading the SGML-DTD file *File*. Same as `load_dtd/3` with empty option list.

load_dtd(*+DTD, +File, +Options*)
Define the DTD by loading *File*. Defined options are the `dialect` option from `open_dtd/3` and the `encoding` option from `open/4`. Notably the `dialect` option must match the dialect used for subsequent parsing using this DTD.

open_dtd(*+DTD, +Options, -OutStream*)
Open a DTD as an output stream. See `load_dtd/2` for an example. Defined options are:

dialect(*Dialect*)
Define the DTD dialect. Default is `sgml`. Using `xml` or `xmlns` processes the DTD case-sensitive.

dtd(*+DocType, -DTD*)
Find the DTD representing the indicated *doctype*. This predicate uses a cache of DTD objects. If a doctype has no associated dtd, it searches for a file using the file search path `dtd` using the call:

```
...,
absolute_file_name(dtd(Type),
                   [ extensions([dtd]),
                     access(read)
                   ], DtdFile),
...
```

Note that DTD objects may be modified while processing errornous documents. For example, loading an SGML document starting with `<?xml ...?>` switches the DTD to XML mode and encountering unknown elements adds these elements to the DTD object. Re-using a DTD object to parse multiple documents should be restricted to situations where the documents processed are known to be error-free.

dtd_property(*+DTD, ?Property*)
This predicate is used to examine the content of a DTD. Property is one of:

doctype(*DocType*)
An atom representing the document-type defined by this DTD.

elements(*ListOfElements*)
A list of atoms representing the names of the elements in this DTD.

element(*Name, Omit, Content*)
The DTD contains an element with the given name. *Omit* is a term of the format `omit`(*OmitOpen, OmitClose*), where both arguments are booleans (`true` or `false` representing whether the open- or close-tag may be omitted. *Content* is the content-model of the element represented as a Prolog term. This term takes the following form:

empty
The element has no content.

cdata
The element contains non-parsed character data. All data up to the matching end-tag is included in the data (*declared content*).

rcdata
As `cdata`, but entity-references are expanded.

any
The element may contain any number of any element from the DTD in any order.

#pcdata
The element contains parsed character data .

element(*A*)
n element with this name.

*****(*SubModel*)
0 or more appearances.

?(*SubModel*)
0 or one appearance.

+(*SubModel*)
1 or more appearances.

, (*SubModel1, SubModel2*)
SubModel1 followed by *SubModel2*.

&(*SubModel1, SubModel2*)
SubModel1 and *SubModel2* in any order.

|(*SubModel1, SubModel2*)
SubModel1 or *SubModel2*.

attributes(*Element, ListOfAttributes*)
ListOfAttributes is a list of atoms representing the attributes of the element *Element*.

attribute(*Element, Attribute, Type, Default*)
Query an element. *Type* is one of `cdata`, `entity`, `id`, `idref`, `name`, `nmtoken`, `notation`, `number` or `nutoken`. For DTD types that allow for a list, the notation `list`(*Type*) is used. Finally, the DTD construct `(a|b|...)` is mapped to the term `nameof`(*ListOfValues*).

Default describes the sgml default. It is one `required`, `current`, `conref` or `implied`. If a real default is present, it is one of `default`(*Value*) or `fixed`(*Value*).

entities(*ListOfEntities*)
ListOfEntities is a list of atoms representing the names of the defined entities.

entity(*Name, Value*)
Name is the name of an entity with given value. Value is one of

Atom
If the value is atomic, it represents the literal value of the entity.

system(*Url*)
Url is the URL of the system external entity.

public(*Id, Url*)
For external public entities, *Id* is the identifier. If an URL is provided this is returned in *Url*. Otherwise this argument is unbound.

notations(*ListOfNotations*)
Returns a list holding the names of all `NOTATION` declarations.

notation(*Name, Decl*)
Unify *Decl* with a list if `system`(+*File*) and/or `public`(+*PublicId*).

3.4.1 The DOCTYPE declaration

As this parser allows for processing partial documents and process the DTD separately, the DOCTYPE declaration plays a special role.

If a document has no DOCTYPE declaraction, the parser returns a list holding all elements and CDATA found. If the document has a DOCTYPE declaraction, the parser will open the element defined in the DOCTYPE as soon as the first real data is encountered.

3.5 Extracting a DTD

Some documents have no DTD. One of the neat facilities of this library is that it builds a DTD while parsing a document with an *implicit* DTD. The resulting DTD contains all elements encountered in the document. For each element the content model is a disjunction of elements and possibly `#PCDATA` that can be repeated. Thus, if we found element `y` and CDATA in element `x`, the model is:

```
<!ELEMENT x - - (y|#PCDATA)*>
```

Any encountered attribute is added to the attribute list with the type `CDATA` and default `#IMPLIED`.

The example below extracts the elements used in an unknown XML document.

```
elements_in_xml_document(File, Elements) :-
        load_structure(File, _,
                       [ dialect(xml),
                         dtd(DTD)
                       ]),
        dtd_property(DTD, elements(Elements)),
        free_dtd(DTD).
```

3.6 Parsing Primitives

new_sgml_parser(-*Parser, +Options***)**

Creates a new parser. A parser can be used one or multiple times for parsing documents or parts thereof. It may be bound to a DTD or the DTD may be left implicit, in which case it is created from the document prologue or parsing is performed without a DTD. Options:

dtd(*?DTD***)**

If specified with an initialised DTD, this DTD is used for parsing the document, regardless of the document prologue. If specified using as a variable, a reference to the created DTD is returned. This DTD may be created from the document prologue or build implicitely from the document's content.

free_sgml_parser(+*Parser***)**

Destroy all resources related to the parser. This does not destroy the DTD if the parser was created using the `dtd(`*DTD*`)` option.

set_sgml_parser(+*Parser, +Option***)**

Sets attributes to the parser. Currently defined attributes:

file(*File*)
Sets the file for reporting errors and warnings. Sets the line to 1.

line(*Line*)
Sets the current line. Useful if the stream is not at the start of the (file) object for generating proper line-numbers.

charpos(*Offset*)
Sets the current character location. See also the `file`(*File*) option.

dialect(*Dialect*)
Set the markup dialect. Known dialects:

sgml
The default dialect is to process as SGML. This implies markup is case-insensitive and standard SGML abbreviation is allowed (abreviated attributes and omitted tags).

xml
This dialect is selected automatically if the processing instruction `<?xml ...>` is encountered. See section 3.3 for details.

xmlns
Process file as XML file with namespace support. See section 3.3.1 for details. See also the `qualify_attributes` option below.

xmlns(+*URI*)
Set the default namespace of the outer environment. This option is provided to process partial XML content with proper namespace resolution.

xmlns(+*NS, +URI*)
Specify a namespace for the outer environment. This option is provided to process partial XML content with proper namespace resolution.

qualify_attributes(*Boolean*)
How to handle unqualified attribute (i.e. without an explicit namespace) in XML namespace (`xmlns`) mode. Default and standard compliant is not to qualify such elements. If `true`, such attributes are qualified with the namespace of the element they appear in. This option is for backward compatibility as this is the behaviour of older versions. In addition, the namespace document suggests unqualified attributes are often interpreted in the namespace of their element.

space(*SpaceMode*)
Define the initial handling of white-space in PCDATA. This attribute is described in section 3.2.

number(*NumberMode*)
If `token` (default), attributes of type number are passed as a Prolog atom. If `integer`, such attributes are translated into Prolog integers. If the conversion fails (e.g. due to overflow) a warning is issued and the value is passed as an atom.

encoding(*Encoding*)
Set the initial encoding. The default initial encoding for XML documents is UTF-8 and for SGML documents ISO-8859-1. XML documents may change the encoding using the `encoding=` attribute in the header. Explicit use of this option is only required to parse non-conforming documents. Currently accepted values are `iso-8859-1` and `utf-8`.

doctype(*Element*)

Defines the toplevel element expected. If a `<!DOCTYPE` declaration has been parsed, the default is the defined doctype. The parser can be instructed to accept the first element encountered as the toplevel using `doctype(_)`. This feature is especially useful when parsing part of a document (see the `parse` option to `sgml_parse/2`.

get_sgml_parser(+*Parser, -Option*)

Retrieve infomation on the current status of the parser. Notably useful if the parser is used in the call-back mode. Currently defined options:

file(-*File*)

Current file-name. Note that this may be different from the provided file if an external entity is being loaded.

line(-*Line*)

Line-offset from where the parser started its processing in the file-object.

charpos(-*CharPos*)

Offset from where the parser started its processing in the file-object. See section 6.

charpos(-*Start, -End*)

Character offsets of the start and end of the source processed causing the current call-back. Used in `PceEmacs` to for colouring text in SGML and XML modes.

source(-*Stream*)

Prolog stream being processed. May be used in the `on_begin`, *etc.* callbacks from `sgml_parse/2`.

dialect(-*Dialect*)

Return the current dialect used by the parser (`sgml`, `xml` or `xmlns`).

event_class(-*Class*)

The *event class* can be requested in call-back events. It denotes the cause of the event, providing useful information for syntax highlighting. Defined values are:

explicit

The code generating this event is explicitely present in the document.

omitted

The current event is caused by the insertion of an omitted tag. This may be a normal event in SGML mode or an error in XML mode.

shorttag

The current event (`begin` or `end`) is caused by an element written down using the *shorttag* notation (`<tag/value/>`.

shortref

The current event is caused by the expansion of a *shortref*. This allows for highlighting shortref strings in the source-text.

doctype(-*Element*)

Return the defined document-type (= toplevel element). See also `set_sgml_parser/2`.

dtd(-*DTD*)

Return the currently used DTD. See `dtd_property/2` for obtaining information on the DTD such as element and attribute properties.

context(-*StackOfElements*)

Returns the stack of currently open elements as a list. The head of this list is the current element. This can be used to determine the context of, for example, CDATA events in call-back mode. The elements are passed as atoms. Currently no access to the attributes is provided.

allowed(-*Elements*)

Determines which elements may be inserted at the current location. This information is returned as a list of element-names. If character data is allowed in the current location, `#pcdata` is part of *Elements*. If no element is open, the *doctype* is returned.

This option is intended to support syntax-sensitive editors. Such an editor should load the DTD, find an appropriate starting point and then feed all data between the starting point and the caret into the parser. Next it can use this option to determine the elements allowed at this point. Below is a code fragment illustrating this use given a parser with loaded DTD, an input stream and a start-location.

```
...,
seek(In, Start, bof, _),
set_sgml_parser(Parser, charpos(Start)),
set_sgml_parser(Parser, doctype(_)),
Len is Caret - Start,
sgml_parse(Parser,
           [ source(In),
             content_length(Len),
             parse(input)       % do not complete document
           ]),
get_sgml_parser(Parser, allowed(Allowed)),
...
```

sgml_parse(+*Parser, +Options*)

Parse an XML file. The parser can operate in two input and two output modes. Output is either a structured term as described with `load_structure/2` or call-backs on predefined events. The first is especially suitable for manipulating not-too-large documents, while the latter provides a primitive means for handling very large documents.

Input is a stream. A full description of the option-list is below.

document(-*Term*)

A variable that will be unified with a list describing the content of the document (see `load_structure/2`).

source(+*Stream*)

An input stream that is read. This option *must* be given.

content_length(+*Characters*)

Stop parsing after *Characters*. This option is useful to parse input embedded in *envelopes*, such as the HTTP protocol.

parse(*Unit*)

Defines how much of the input is parsed. This option is used to parse only parts of a file.

file
Default. Parse everything upto the end of the input.

element
The parser stops after reading the first element. Using `source`(*Stream*), this implies reading is stopped as soon as the element is complete, and another call may be issued on the same stream to read the next element.

content
The value `content` is like `element` but assumes the element has already been opened. It may be used in a call-back from `call`(`on_begin`, *Pred*) to parse individual elements after validating their headers.

declaration
This may be used to stop the parser after reading the first declaration. This is especially useful to parse only the `doctype` declaration.

input
This option is intended to be used in conjunction with the `allowed`(*Elements*) option of `get_sgml_parser/2`. It disables the parser's default to complete the parse-tree by closing all open elements.

max_errors(+*MaxErrors*)
Set the maximum number of errors. If this number is exceeded further writes to the stream will yield an I/O error exception. Printing of errors is suppressed after reaching this value. The default is 100.

syntax_errors(+*ErrorMode*)
Defines how syntax errors are handled.

quiet
Suppress all messages.

print
Default. Pass messages to `print_message/2`.

style
Print dubious input such as attempts for redefinitions in the DTD using `print_message/2` with severity `informational`.

xml_no_ns(+*Mode*)
Error handling if an XML namespace is not defined. Default generates an error. If `quiet`, the error is suppressed. Can be used together with `call`(*urlns, Closure*) to provide external expansion of namespaces. See also section 3.3.1.

call(+*Event, :PredicateName*)
Issue call-backs on the specified events. *PredicateName* is the name of the predicate to call on this event, possibly prefixed with a module identifier. If the handler throws an exception, parsing is stopped and `sgml_parse/2` re-throws the exception. The defined events are:

begin
An open-tag has been parsed. The named handler is called with three arguments: *Handler*(+*Tag, +Attributes, +Parser*).

end
A close-tag has been parsed. The named handler is called with two arguments: *Handler*(+*Tag, +Parser*).

cdata
CDATA has been parsed. The named handler is called with two arguments: `Handler`(+*CDATA, +Parser*), where CDATA is an atom representing the data.

pi
A processing instruction has been parsed. The named handler is called with two arguments: *Handler*(+*Text, +Parser*), where *Text* is the text of the processing instruction.

decl
A declaration (`<!...>`) has been read. The named handler is called with two arguments: *Handler*(+*Text, +Parser*), where *Text* is the text of the declaration with comments removed.
This option is expecially useful for highlighting declarations and comments in editor support, where the location of the declaration is extracted using `get_sgml_parser/2`.

error
An error has been encountered. the named handler is called with three arguments: *Handler*(+*Severity, +Message, +Parser*), where *Severity* is one of `warning` or `error` and *Message* is an atom representing the diagnostic message. The location of the error can be determined using `get_sgml_parser/2`
If this option is present, errors and warnings are not reported using `print_message/3`

xmlns
When parsing an in `xmlns` mode, a new namespace declaraction is pushed on the environment. The named handler is called with three arguments: *Handler*(+*NameSpace, +URL, +Parser*). See section 3.3.1 for details.

urlns
When parsing an in `xmlns` mode, this predicate can be used to map a url into either a canonical URL for this namespace or another internal identifier. See section 3.3.1 for details.

3.6.1 Partial Parsing

In some cases, part of a document needs to be parsed. One option is to use `load_structure/2` or one of its variations and extract the desired elements from the returned structure. This is a clean solution, especially on small and medium-sized documents. It however is unsuitable for parsing really big documents. Such documents can only be handled with the call-back output interface realised by the `call`(*Event, Action*) option of `sgml_parse/2`. Event-driven processing is not very natural in Prolog.

The SGML2PL library allows for a mixed approach. Consider the case where we want to process all descriptions from RDF elements in a document. The code below calls `process_rdf_description(Element)` on each element that is directly inside an RDF element.

```
:- dynamic
        in_rdf/0.

load_rdf(File) :-
```

```
        retractall(in_rdf),
        open(File, read, In),
        new_sgml_parser(Parser, []),
        set_sgml_parser(Parser, file(File)),
        set_sgml_parser(Parser, dialect(xml)),
        sgml_parse(Parser,
                   [ source(In),
                     call(begin, on_begin),
                     call(end, on_end)
                   ]),
        close(In).

on_end('RDF', _) :-
        retractall(in_rdf).

on_begin('RDF', _, _) :-
        assert(in_rdf).
on_begin(Tag, Attr, Parser) :-
        in_rdf, !,
        sgml_parse(Parser,
                   [ document(Content),
                     parse(content)
                   ]),
        process_rdf_description(element(Tag, Attr, Content)).
```

3.7 Type checking

xml_is_dom(@*Term*)
: True if *Term* is an SGML/XML term as produced by one of the above predciates and acceptable by `xml_write/3` and friends.

4 Stream encoding issues

The parser can deal with ISO Latin-1 and UTF-8 encoded files, doing decoding based on the encoding argument provided to `set_sgml_parser/2` or, for XML, based on the `encoding` attribute of the XML header. The parser reads from SWI-Prolog streams, which also provide encoding handling. Therefore, there are two modes for parsing. If the SWI-Prolog stream has encoding `octet` (which is the default for binary streams), the decoder of the SGML parser will be used and positions reported by the parser are octet offsets in the stream. In other cases, the Prolog stream decoder is used and offsets are character code counts.

5 library(xpath): Select nodes in an XML DOM

See also `http://www.w3.org/TR/xpath`

The library `xpath.pl` provides predicates to select nodes from an XML DOM tree as produced by library(sgml) based on descriptions inspired by the XPATH language.

The predicate `xpath/3` selects a sub-structure of the DOM non-deterministically based on an xpath-like specification. Not all selectors of XPATH are implemented, but the ability to mix `xpath/3` calls with arbitrary Prolog code provides a powerful tool for extracting information from XML parse-trees.

xpath_chk(+*DOM, +Spec, ?Content*) *[semidet]*

Semi-deterministic version of `xpath/3`.

xpath(+*DOM, +Spec, ?Content*) *[nondet]*

Match an element in a *DOM* structure. The syntax is inspired by XPath, using () rather than [] to select inside an element. First we can construct paths using / and //:

`//`**Term** Select any node in the *DOM* matching term.

`/`**Term** Match the root against Term.

Term Select the immediate children of the root matching Term.

The Terms above are of type *callable*. The functor specifies the element name. The element name '*' refers to any element. The name `self` refers to the top-element itself and is often used for processing matches of an earlier `xpath/3` query. A term NS:Term refers to an XML name in the namespace NS. Optional arguments specify additional constraints and functions. The arguments are processed from left to right. Defined conditional argument values are:

Integer The N-th element with the given name

`last` The last element with the given name.

`last` **- IntExpr** The IntExpr-th element counting from the last (0-based)

Defined function argument values are:

`self` Evaluate to the entire element

`text` Evaluates to all text from the sub-tree as an atom

`normalize_space` As `text`, but uses `normalize_space/2` to normalise white-space in the output

`number` Extract an integer or float from the value. Ignores leading and trailing white-space

`@`**Attribute** Evaluates to the value of the given attribute

In addition, the argument-list can be *conditions*:

Left = Right Succeeds if the left-hand unifies with the right-hand. E.g. normalize_space = 'euro'

contains(Haystack, Needle) Succeeds if Needle is a sub-string of Haystack.

Examples:

Match each table-row in *DOM*:

```
xpath(DOM, //tr, TR)
```

Match the last cell of each tablerow in *DOM*. This example illustrates that a result can be the input of subsequent `xpath/3` queries. Using multiple queries on the intermediate TR term guarantee that all results come from the same table-row:

```
xpath(DOM, //tr, TR),
xpath(TR,  /td(last), TD)
```

Match each `href` attribute in an <a> element

```
xpath(DOM, //a(@href), HREF)
```

Suppose we have a table containing rows where each first column is the name of a product with a link to details and the second is the price (a number). The following predicate matches the name, URL and price:

```
product(DOM, Name, URL, Price) :-
    xpath(DOM, //tr, TR),
    xpath(TR, td(1), C1),
    xpath(C1, /self(normalize_space), Name),
    xpath(C1, a(@href), URL),
    xpath(TR, td(2, number), Price).
```

Suppose we want to select books with genre="thriller" from a tree containing elements `<book genre=...>`

```
thriller(DOM, Book) :-
    xpath(DOM, //book(@genre=thiller), Book).
```

6 Processing Indexed Files

In some cases applications wish to process small portions of large SGML, XML or RDF files. For example, the *OpenDirectory* project by Netscape has produced a 90MB RDF file representing the main index. The parser described here can process this document as a unit, but loading takes 85 seconds on a Pentium-II 450 and the resulting term requires about 70MB global stack. One option is to process the entire document and output it as a Prolog fact-base of RDF triplets, but in many cases this is undesirable. Another example is a large SGML file containing online documentation. The application normally wishes to provide only small portions at a time to the user. Loading the entire document into memory is then undesirable.

Using the `parse`(*element*) option, we open a file, seek (using `seek/4`) to the position of the element and read the desired element.

The index can be built using the call-back interface of `sgml_parse/2`. For example, the following code makes an index of the `structure.rdf` file of the OpenDirectory project:

```
:- dynamic
        location/3.                     % Id, File, Offset

rdf_index(File) :-
        retractall(location(_,_)),
        open(File, read, In, [type(binary)]),
        new_sgml_parser(Parser, []),
        set_sgml_parser(Parser, file(File)),
        set_sgml_parser(Parser, dialect(xml)),
        sgml_parse(Parser,
                   [ source(In),
                     call(begin, index_on_begin)
                   ]),
        close(In).

index_on_begin(_Element, Attributes, Parser) :-
        memberchk('r:id'=Id, Attributes),
        get_sgml_parser(Parser, charpos(Offset)),
        get_sgml_parser(Parser, file(File)),
        assert(location(Id, File, Offset)).
```

The following code extracts the RDF element with required id:

```
rdf_element(Id, Term) :-
        location(Id, File, Offset),
        load_structure(File, Term,
                       [ dialect(xml),
                         offset(Offset),
                         parse(element)
                       ]).
```

7 External entities

While processing an SGML document the document may refer to external data. This occurs in three places: external parameter entities, normal external entities and the `DOCTYPE` declaration. The current version of this tool deals rather primitively with external data. External entities can only be loaded from a file and the mapping between the entity names and the file is done using a *catalog* file in a format compatible with that used by James Clark's SP Parser, based on the SGML Open (now OASIS) specification.

Catalog files can be specified using two primitives: the predicate `sgml_register_catalog_file/2` or the environment variable `SGML_CATALOG_FILES` (compatible with the SP package).

sgml_register_catalog_file(+*File, +Location*)

Register the indicated *File* as a catalog file. *Location* is either `start` or `end` and defines whether the catalog is considered first or last. This predicate has no effect if *File* is already part of the catalog.

If no files are registered using this predicate, the first query on the catalog examines `SGML_CATALOG_FILES` and fills the catalog with all files in this path.

Two types of lines are used by this package.

`DOCTYPE` *doctype file*
`PUBLIC` **"***Id***"** *file*

The specified *file* path is taken relative to the location of the catolog file. For the `DOCTYPE` declaraction, `sgml` first makes an attempt to resolve the `SYSTEM` or `PUBLIC` identifier. If this fails it tries to resolve the *doctype* using the provided catalog files.

Strictly speaking, `sgml` breaks the rules for XML, where system identifiers must be Universal Resource Indicators, not local file names. Simple uses of relative URIs will work correctly under UNIX and Windows.

In the future we will design a call-back mechanism for locating and processing external entities, so Prolog-based file-location and Prolog resources can be used to store external entities.

8 library(pwp): Prolog Well-formed Pages

author Richard O'Keefe
To be done Support compilation of PWP input files

PWP is an approach to server-side scripting using Prolog which is based on a simple key principle:

- The source form of a PWP should be WELL-FORMED XML

Especially when generating XML rather than HTML, this is such an obvious thing to do. We have many kinds of XML checking tools.

- We can tell whether an XML document is WELL FORMED (all the punctuation is right, all tags balance) using practically any decent parser, including SWI Prolog's 'sgml'.
- If we can write a Document Type Description then we can check that a document is VALID using tools like Open SP (formerly James Clark's SP) or SWI Prolog's 'sgml'. This does not guarantee that the output will be valid, but it does catch a lot of errors very early.
- If we can write an XML Schema then we can check that a document is schema-valid. (SWI Prolog's 'sgml' does not yet come with a schema validator, but who knows what the future holds?).
- Since an XML document is just a data structure, we can use any checking tool that we can write in Prolog, IF the input is well-formed so that we can load a template as a Prolog data structure.

Having decided that the input should be well formed, that means **NO NEW SYNTAX**

None of the weird and horrible <% ... %> or whatever not-quite-XML stuff you see in other template systems, making checking so very hard (and therefore, making errors so distressingly common).

That in turns means that PWP "markup" must be based on special elements or special attributes. The fact that an XML parser must allow undeclared attributes on any element even when validating, but must not allow undeclared elements, suggests doing this through attributes. In particular, one should be able to take an existing DTD, such as an XHTML DTD, and just use that without modification. So the design reduces to

- Allow dynamic data selection, insertion, and transformation just using a small number of extra attributes.

This description uses the following name space:

```
xmlns:pwp='http://www.cs.otago.ac.nz/staffpriv/ok/pwp.pl'
```

The attributes are

- pwp:ask = Query
- pwp:use = Term
- pwp:how = text | xml
- pwp:tag = QName or '-'
- pwp:att = '' | 'one non-alphanumeric character'

Here's what they mean. Each element is expanded in the context of a set of variable bindings. After expansion, if the tag is not mapped to '-', all attributes in the pwp: namespace are removed and the children elements are recursively expanded.

- pwp:ask = Query
  - Query is a Prolog goal. For each solution of Query, the element is further processed with the new variables of Query added to the context.
  - If Query is not a well formed Prolog goal, or if execution of Query throws an exception, page transformation comes to a complete halt and no page is generated.
- pwp:use = Term
- pwp:how = text | xml | text-file | xml-file

  Term is a Prolog term; variables in Term are bound by the context. An empty Term is regarded as a missing value for this attribute. The Prolog variable CONTEXT refers to the entire context, a list of Name = Value, where Name is a Prolog atom holding the name of the context variable and Value is an arbitrary Prolog term.

  - If pwp:how is text, The value of Term is used to define a sequence of characters.
    - A number produces the same characters that `write/1` would.
    - An atom produces the same characters that `write/1` would.

* A string produces the same characters that `write/1` would.
* A list of character codes produces those characters.
* The following terms produce the same sequence of characters that the corresponding goal would have sent to the current output stream:

  write(*Datum***)**

  writeq(*Datum***)**

  write_canonical(*Datum***)**

  print(*Datum***)**

  print(*Datum***)**

  format(*Format***)**

  format(*Format, Arguments***)**

* A singleton list [X] defines the characters that X defines.
* Any other term F(T1,...,Tn) defines the characters that T1 defines, followed by the characters that T2 defines, ..., followed by the characters that Tn defines.

– If pwp:how is xml,

  The value of Term must be an XML term as defined in the SGML2PL documentation or a list of such terms. A single term is taken as if it had been [Term]. The resulting list of terms replaces the children of the current element and will not be further processed.

– If pwp:how is text-file,

  The value of Term is used to define a sequence of characters. That sequence of characters is used as a file name. The file is read as a sequence of characters, and that sequence used as character data.

– If pwp:how is xml-file,

  The value of Term is used to define a sequence of characters. That sequence of characters is used as a file name. The file is loaded as XML, and the sequence of XML items thus obtained used. This means that PWP provides XML inclusion without depending on the parser to support XInclude.

The default value for pwp:how is text.

- pwp:tag = QName or '-'

  If pwp:tag is missing or the value is empty, the current element appears in the output (after further processing) with its present tag. If pwp:tag is a QName, the current element appears (...) with that as its tag. That option is most useful in DTDs, where an "authoring" DTD may use one tag and have it automatically mapped to another

tag in the output, e.g., <item> -> . Finally, if pwp:tag is '-', the children of the current element (either the result of pwp:use or the transformed original children, whichever applies) appear in the output but there is no element around them.

A missing or empty pwp:ask is just like pwp:ask = 'true'.

- pwp:att = '' | 'one non-alphanumeric character'.

  Attributes in the pwp namespace are not affected by this attribute. Such attributes are always stripped out and never substituted into.

  If pwp:att is missing or empty, attributes of the current element are copied over to the output unchanged.

  If pwp:att = 'c' for some non-alphanumeric character c, each attribute is examined for occurrences of c(...)c and c[...]c which are as short as possible. There is no one character which could be used every time, so you have to explicitly choose a substitution marker which is safe for the data you do not want to be altered. None of the pwp attributes are inherited, least of all this one.

  Text outside c(...)c groups is copied unchanged; text inside a c(...)c group is parsed as a Prolog term and treated as if by pwp:how = text. Text inside a c[...]c group is evaluated (in the current context), and if it fails, the entire attribute will be removed from the element.

Examples:

1. **A "Hello World" like example**

```
<html
  xmlns:pwp="http://www.cs.otago.ac.nz/staffpriv/ok/pwp.pl"
  pwp:ask = "ensure_loaded(msg), once(msg(Greeting))">
  <head>
    <title pwp:use="Greeting"/>
  </head>
  <body>
    <p><span pwp:use="Greeting" pwp:tag='-'/></p>
  </body>
</html>
```

where `msg.pl` contains

```
msg('Hello, World!').
```

This example illustrates an important point. Prolog Well-Formed Pages provide **NO** way to physically incorporate Prolog **clauses** into a page template. Prolog clauses must be put in separate files which can be checked by a Prolog syntax checker, compiler, cross-referencer, &c WITHOUT the Prolog tool in question needing to know anything whatsoever about PWP. You load the files using pwp:ask on the root element.

2. **Binding some variables and using them**

```
<html
  xmlns:pwp="http://www.cs.otago.ac.nz/staffpriv/ok/pwp.pl">
  <head><title>Example 2</title></head>
  <body pwp:ask="Hello = 'Hello world', A = 20, B = 22">
    <h1 pwp:use="Hello"/>
    <p>The answer is <span pwp:use="C" pwp:ask="C is A+B"/>.</p>
  </body>
</html>
```

3. **Making a table** We are given a Prolog database `staff.pl` defining staff(NickName, Full-Name, Office, Phone, E_Mail_Address). status(NickName, full_time | part_time). We want to make a phone list of full time staff.

```
<html
  xmlns:pwp="http://www.cs.otago.ac.nz/staffpriv/ok/pwp.pl"
  pwp:ask='ensure_loaded(staff)'>
  <head>
    <title>Phone list for Full-Time staff.</title>
  </head>
  <body>
    <h1>Phone list for Full-Time staff.</h1>
    <table
      pwp:ask = "setof(FullName-Phone,
                       N^O^E^(
                         status(N, full_time),
                         staff(N, FullName, O, Phone, E)
                       ),
                       Staff_List)">
      <tr><th>Name</th><th>Phone</th></tr>
      <tr pwp:ask="member(FullName-Phone, Staff_List)">
        <td pwp:use="FullName"/>
        <td pwp:use="Phone"/>
      </tr>
    </table>
  </body>
</html>
```

4. **Substituting into an attribute** Same data base as before, but now we want to make a mailing list page.

```
<html
  xmlns:pwp="http://www.cs.otago.ac.nz/staffpriv/ok/pwp.pl"
  pwp:ask='ensure_loaded(staff)'>
  <head>
```

```
    <title>Phone list for Full-Time staff.</title>
  </head>
  <body>
    <h1>Phone list for Full-Time staff.</h1>
    <table
      pwp:ask = "setof(FullName-E_Mail,
                       N^O^P^staff(N, FullName, O, P, E_Mail),
                       Staff_List)">
      <tr><th>Name</th><th>Address</th></tr>
      <tr pwp:ask="member(FullName-E_Mail, Staff_List)">
        <td pwp:use="FullName"/>
        <td><a pwp:use="E_Mail"
               pwp:att='$' href="mailto:$(E_Mail)$"/></td>
      </tr>
    </table>
  </body>
</html>
```

5. **If-then-else effect** A page that displays the value of the 'SHELL' environment variable if it has one, otherwise displays 'There is no default shell.'

```
<html
  xmlns:pwp="http://www.cs.otago.ac.nz/staffpriv/ok/pwp.pl">
  <head><title>$SHELL</title></head>
  <body>
    <p pwp:ask="getenv('SHELL', Shell)"
    >The default shell is <span pwp:tag="-" pwp:use="Shell"/>.</p>
    <p pwp:ask="\+getenv('SHELL',_)">There is no default shell.</p>
  </body>
</html>
```

There is one other criterion for a good server-side template language:

It should be possible to compile templates so as to eliminate most if not all interpretation overhead.

This particular notation satisfies that criterion with the limitation that the conversion of a term to character data requires run-time traversal of terms (because the terms are not known until run time).

pwp_files(*:In:atom, +Out:atom*) *[det]*

loads an Xml document from the file named *In*, transforms it using the PWP attributes, and writes the transformed version to the new file named *Out*.

pwp_stream(*:Input:input_stream, +Output:output_stream, +Context:list*) *[det]*

Loads an Xml document from the given *Input* stream, transforms it using the PWP attributes,

and writes the transformed version to the given *Output* stream. *Context* provides initial contextual variables and is a list of Name=Value.

pwp_xml(*:In:list(xml), -Out:list(xml), +Context*)
maps down a list of XML items, acting specially on elements and copying everything else unchanged, including white space. The *Context* is a list of 'VariableName'=CurrentValue bindings.

9 Writing markup

9.1 Writing documents

The library `sgml_write` provides the inverse of the parser, converting the parser's output back into a file. This process is fairly simple for XML, but due to the power of the SGML DTD it is much harder to achieve a reasonable generic result for SGML.

These predicates can write the output in two encoding schemas depending on the encoding of the *Stream*. In UTF-8 mode, all characters are encoded using UTF-8 sequences. In ISO Latin-1 mode, characters outside the ISO Latin-1 range are represented using a named character entity if provided by the DTD or a numeric character entity.

xml_write(*+Stream, +Term, +Options*)
Write the XML header with encoding information and the content of the document as represented by *Term* to *Stream*. This predicate deals with XML with or without namespaces. If namespace identifiers are not provided they are generated. This predicate defines the following *Options*

dtd(*DTD*)
Specify the DTD. In SGML documents the DTD is required to distinguish between elements that are declared empty in the DTD and elements that just happen to have no content. Further optimisation (shortref, omitted tags, etc.) could be considered in the future. The DTD is also used to find the declared named character entities.

doctype(*Doctype*)
Document type to include in the header. When omitted it is taken from the outer element.

header(*Bool*)
If *Bool* is `false`, the XML header is suppressed. Useful for embedding in other XML streams.

layout(*Bool*)
Do/do not emit layout characters to make the output readable, Default is to emit layout. With layout enabled, elements only containing other elements are written using increasing indentation. This introduces (depending on the mode and defined whitespace handling) CDATA sequences with only layout between elements when read back in. If `false`, no layout characters are added. As this mode does not need to analyse the document it is faster and guarantees correct output when read back. Unfortunately the output is hardly human readable and causes problems with many editors.

indent(*Integer*)
Set the initial element indentation. It more than zero, the indent is written before the document.

nsmap(*Map*)
Set the initial namespace map. *Map* is a list of *Name* = *URI*. This option, together with `header` and `ident` is added to use `xml_write/3` to generate XML that is embedded in a larger XML document.

net(*Bool*)
Use/do not use *Null End Tags*. For XML, this applies only to empty elements, so you get `<foo/>` (default, `net`(*true*)) or `\bnfmeta{foo}</foo>` (`net`(*false*)). For SGML, this applies to empty elements, so you get `\bnfmeta{foo}` (if foo is declared to be `EMPTY` in the DTD), `\bnfmeta{foo}</foo>` (default, `net`(*false*)) or `<foo//` (`net`(*true*)). In SGML code, short character content not containing `/` can be emitted as `\bnfmeta{b}xxx</b>` (default, `net`(*false*) or `<b/xxx/` (`net`(*true*))

sgml_write(+*Stream, +Term, +Options*)
Write the SGML `DOCTYPE` header and the content of the document as represented by *Term* to *Stream*. The *Options* are described with `xml_write/3`.

html_write(+*Stream, +Term, +Options*)
Same as `sgml_write/3`, but passes the HTML DTD as obtained from `dtd/2`. The *Options* are described with `xml_write/3`.

9.2 XML Quote primitives

In most cases, the preferred way to create an XML document is to create a Prolog tree of `element`(*Name, Attributes, Content*) terms and call `xml_write/3` to write this to a stream. There are some exceptions where one might not want to pay the price of the intermediate representation. For these cases, this library contains building blocks for emitting markup data. The quote funtions return a version of the input text into one that contains entities for characters that need to be escaped. These are the XML meta characters and the characters that cannot be expressed by the document encoding. Therefore these predicates accept an *encoding* argument. Accepted values are `ascii`, `iso_latin_1`, `utf8` and `unicode`. Versions with two arguments are provided for backward compatibility, making the safe `ascii` encoding assumption.

xml_quote_attribute(+*In, -Quoted, +Encoding*)
Map the characters that may not appear in XML attributes to entities. Currently these are `<>&&quot;`.[3] Characters that cannot represented in *Encoding* are mapped to XML character entities.

xml_quote_attribute(+*In, -Quoted*)
Backward compatibility version for `xml_quote_attribute/3`. Assumes `ascii` encoding.

xml_quote_cdata(+*In, -Quoted, +Encoding*)
Very similar to `xml_quote_attribute/3`, but does not quote the single- and double-quotes.

xml_quote_cdata(+*In, -Quoted*)
Backward compatibility version for `xml_quote_cdata/3`. Assumes `ascii` encoding.

xml_name(+*In, +Encoding*)
Succeed if *In* is an atom or string that satisfies the rules for a valid XML element or

[3] Older versions also mapped `'` to `&apos;`.

attribute name. As with the other predicates in this group, if *Encoding* cannot represent one of the characters, this function fails. Character classification is based on `http://www.w3.org/TR/2006/REC-xml-20060816`.

xml_name(+*In*)
Backward compatibility version for `xml_name/2`. Assumes `ascii` encoding.

10 Unsupported features

The current parser is rather limited. While it is able to deal with many serious documents, it omits several less-used features of SGML and XML. Known missing SGML features include

- *NOTATION on entities*
  Though notation is parsed, notation attributes on external entity declarations are not handed to the user.

- *NOTATION attributes*
  SGML notations may have attributes, declared using `<!ATTLIST #NOTATION name attributes>`. Those data attributes are provided when you declare an external CDATA, NDATA, or SDATA entity.

  XML does not include external CDATA, NDATA, or SDATA entities, nor any of the other uses to which data attributes are put in SGML, so it doesn't include data attributes for notations either.

  Sgml2pl does not support this feature and is unlikely to; you should be aware that SGML documents using this feature cannot be converted faithfully to XML.

- *SHORTTAG*
  The SGML SHORTTAG syntax is only partially implemented. Currently, `<tag/content/` is a valid abbreviation for `\bnfmeta{tag}content</tag>`, which can also be written as `\bnfmeta{tag}content</>`. Empty start tags (<>), unclosed start tags (`<a<b</verb>`) and unclosed end tags (`\bnfmeta{verb}</a<b`) are not supported.

- *SGML declaration*
  The 'SGML declaration' is fixed, though most of the parameters are handled through indirections in the implementation.

- *The DATATAG feature*
  It is regarded as superseeded by SHORTREF, which is supported. (SP does not support it either.)

- *The RANK feature*
  It is regarded as obsolete.

- *The LINK feature*
  It is regarded as too complicated.

- *The CONCUR feature*
  Concurrent markup allows a document to be tagged according to more than one DTD at the same time. It is not supported.

In XML mode the parser recognises SGML constructs that are not allowed in XML. Also various extensions of XML over SGML are not yet realised. In particular, XInclude is not implemented because the designers of XInclude can't make up their minds whether to base it on elements or attributes yet, let alone details.

11 Installation

11.1 Unix systems

Installation on Unix system uses the commonly found `configure`, `make` and `make install` sequence. SWI-Prolog should be installed before building this package. If SWI-Prolog is not installed as `pl`, the environment variable `PL` must be set to the name of the SWI-Prolog executable. Installation is now accomplished using:

```
% ./configure
% make
% make install
```

This installs the foreign libraries in `$PLBASE/lib/$PLARCH` and the Prolog library files in `$PLBASE/library`, where `$PLBASE` refers to the SWI-Prolog 'home-directory'.

12 Acknowledgements

The Prolog representation for parsed documents is based on the SWI-Prolog interface to SP by Anjo Anjewierden.

Richard O'Keefe has put a lot of effort testing and providing bug reports consisting of an illustrative example and explanation of the standard. He also made many suggestions for improving this document.