SWI-Prolog SGML/XML parser

Version 1.0.9, August 2000

Jan Wielemaker

SWI,
University of Amsterdam
The Netherlands
E-mail: jan@swi.psy.uva.nl

Abstract

Markup languages are an increasingly important method for data-representation and exchange. This article documents the package **sgml2pl**, 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 50 kbytes executable on a Pentium or 80 kbytes on a SPARC), 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>

<p>Paragraphs in HTML need not be closed.

<p>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 returned as part of `CDATA`, unless they cannot be represented. See **load_sgml_file/2** for details.

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

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

load_html_file(*+File, -Content***)**
    Load *File* and parse as HTML. Implemented as:

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

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(_+File, -ListOfContent, +Options_**)**
Load the XML file *File* and return the resulting structure in *ListOfContent*. *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 that appeared in the source. No information is returned on other attributes, such as `fixed` or `default` attributes. See **dtd_property/2** for accessing the DTD for this information. 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.

entity(_Code_**)**
If a character-entity (e.g. `Α`) is encoutered that cannot be represented in the Prolog character set, this term is returned, representing the referred character code.

entity(_Name_**)**
If an entity refers to a character-entity holding a single character, but this character cannot be represented in the Prolog character set, this term is returned. For

example, the HTML input text Α < Β is returned as below. Please note that entity names are case sensitive in both SGML and XML.

```
[ entity('Alpha'), ' < ', entity('Beta') ]
```

This is a special case of entity(Code), intended to handle special symbols by their name rather than character code.

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.

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.

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.

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

3.2 Handling white-space

SGPL2PL 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.[1] 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 ... ?>
```

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

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` (").
- *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:`*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)
                       ]).
```

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_. This predicate is defined using the low-level **open_dtd/3** predicate:

```
load_dtd(DTD, DtdFile) :-
        open_dtd(DTD, [], DtdOut),
        open(DtdFile, read, DtdIn),
        copy_stream_data(DtdIn, DtdOut),
        close(DtdIn),
        close(DtdOut).
```

open_dtd(_+DTD, +Options, -OutStream_**)**
: Open a DTD as an output stream. The option-list is currently empty. See **load_dtd/2** for an example.

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),
...
```

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
An 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, File***)**
: Yields the declared file for from a `NOTATION` declaration.

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.

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.

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.

charpos(*-CharPos***)**
Offset from where the parser started its processing in the file-object. See section 4.

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,
   [ goal(feed(In, Len)),
     parse(input) % do not complete document
   ]),
      get_sgml_parser(Parser, allowed(Allowed)),
...

feed(In, Len, Parser) :-
copy_stream_data(In, Parser, Len).
```

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 either a stream or an goal that pushes characters into the parser. 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. Either this option or the `goal(Goal)` option must be provided.

goal(*+Goal* **)**

Goal is a callable term. The predicate **sgml_parse/2** opens an output stream to the parser and invokes `call(Goal, Stream)`, where *Goal* should write the data to be parsed to *Stream*. This option is not compatible with `parse(element)`. This option can be used, for example, to parse a Prolog atom:

```
parse_atom(Atom, Term) :-
        new_sgml_parser(Parser, []),
        sgml_parse(Parser,
                   [ document(Term),
                     goal(provide_atom(Atom))
                   ]),
        free_sgml_parser(Parser).

provide_atom(Atom, ParserStream) :-
        write(ParserStream, Atom).
```

For example:

```
?- parse_atom('<h1>hello world</h1>', X).

X = [element(h1, [], ['hello world'])]
```

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. Using `goal(Goal)` as input, the stream reports an I/O error after completing the first element.

This exception destroyes the built `document(Term)`, making this option useless using 'Goal' driven input.

`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.

print

Default. Pass messages to **print_message/2**.

quiet

Suppress all messages.

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. 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.

entity

An entity that cannot be represented as CDATA has been parsed. The named handler is called with two arguments: *Handler*`(+NameOrCode, +Parser)`.

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)).
```

4 Processing Indexed Files

In some cases applications which 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)
                       ]).
```

5 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, **sgml2pl** 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, **sgml2pl** 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.

6 Missing functionality

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 fully parsed, notation attributes on external entity declarations are not handed to the user.
- *SHORTTAG*
  The SGML SHORTTAG syntax is only partially implemented. Currently, `<tag/content/` is a valid abbreviation for `<tag>content</tag>`, which can also be written as `<tag>content</>`. Empty start tags (`<>`), unclosed start tags (`<a<b`) and unclosed end tags (`</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.

7 Installation

7.1 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'.

8 Acknowledgements

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

Richard O'Keefe pointed out a number of mistakes in a earlier version of this parser, and made many suggestions for improving this document.

A Summary of Predicates

dtd/2	Find or build a DTD for a document type
dtd_property/2	Query elements, entities and attributes in a DTD
free_dtd/1	Free a DTD object
free_sgml_parser/1	Destroy a parser
get_sgml_parser/2	Get parser options
load_dtd/2	Read DTD information from a file
load_html_file/2	Parse HTML file into Prolog term
load_sgml_file/2	Parse SGML file into Prolog term
load_structure/3	Parse XML/SGML/HTML file into Prolog term
load_xml_file/2	Parse XML file into Prolog term
new_dtd/2	Create a DTD object
new_sgml_parser/2	Create a new parser
open_dtd/3	Open a DTD object as an output stream
set_sgml_parser/2	Set parser options (dialect, source, *etc.*)
sgml_parse/2	Parse the input
sgml_register_catalog_file/2	Register a catalog file

Index