SWI-Prolog RDF parser

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: jan@swi-prolog.org

January 18, 2007

Abstract

RDF (**R**esource **D**escription **F**ormat) is a W3C standard for expressing meta-data about web-resources. It has two representations providing the same semantics. RDF documents are normally transferred as XML documents using the RDF-XML syntax. This format is unsuitable for processing. The parser defined here converts an RDF-XML document into the *triple* notation. The library `rdf_write` creates an RDF/XML document from a list of triples.

Contents

1 Introduction

RDF is a promising standard for representing meta-data about documents on the web as well as exchanging frame-based data (e.g. ontologies). RDF is often associated with 'semantics on the web'. It consists of a formal data-model defined in terms of *triples*. In addition, a *graph* model is defined for visualisation and an XML application is defined for exchange.

'Semantics on the web' is also associated with the Prolog programming language. It is assumed that Prolog is a suitable vehicle to reason with the data expressed in RDF models. Most of the related web-infra structure (e.g. XML parsers, DOM implementations) are defined in Java, Perl, C or C++.

Various routes are available to the Prolog user. Low-level XML parsing is due to its nature best done in C or C++. These languages produce fast code. As XML/SGML are at the basis of most of the other web-related formats we will benefit most here. XML and SGML, being very stable specifications, make fast compiled languages even more attractive.

But what about RDF? RDF-XML is defined in XML, and provided with a Prolog term representing the XML document processing it according to the RDF syntax is quick and easy in Prolog. The alternative, getting yet another library and language attached to the system, is getting less attractive. In this document we explore the suitability of Prolog for processing XML documents in general and into RDF in particular.

2 Parsing RDF in Prolog

We realised an RDF compiler in Prolog on top of the **sgml2pl** package (providing a name-space sensitive XML parser). The transformation is realised in two passes.

The first pass rewrites the XML term into a Prolog term conveying the same information in a more friendly manner. This transformation is defined in a high-level pattern matching language defined on top of Prolog with properties similar to DCG (Definite Clause Grammar).

The source of this translation is very close to the BNF notation used by the specification, so correctness is 'obvious'. Below is a part of the definition for RDF containers. Note that XML elements are represented using a term of the format:

`element`(*Name, [AttrName = Value...], [Content ...]*)

```
memberElt(LI) ::=
        \referencedItem(LI).
memberElt(LI) ::=
        \inlineItem(LI).

referencedItem(LI) ::=
        element(\rdf(li),
                [ \resourceAttr(LI) ],
                []).

inlineItem(literal(LI)) ::=
        element(\rdf(li),
                [ \parseLiteral ],
                LI).
inlineItem(description(description, _, _, Properties)) ::=
```

```
        element(\rdf(li),
                [ \parseResource ],
                \propertyElts(Properties)).
inlineItem(LI) ::=
        element(\rdf(li),
                [],
                [\rdf_object(LI)]), !.  % inlined object
inlineItem(literal(LI)) ::=
        element(\rdf(li),
                [],
                [LI]).                  % string value
```

Expression in the rule that are prefixed by the \ operator acts as invocation of another rule-set. The body-term is converted into a term where all rule-references are replaced by variables. The resulting term is matched and translation of the arguments is achieved by calling the appropriate rule. Below is the Prolog code for the **referencedItem** rule:

```
referencedItem(A, element(B, [C], [])) :-
        rdf(li, B),
        resourceAttr(A, C).
```

Additional code can be added using a notation close to the Prolog DCG notation. Here is the rule for a description, producing properties both using **propAttrs** and **propertyElts**.

```
description(description, About, BagID, Properties) ::=
        element(\rdf('Description'),
                \attrs([ \?idAboutAttr(About),
                         \?bagIdAttr(BagID)
                       | \propAttrs(PropAttrs)
                       ]),
                \propertyElts(PropElts)),
        { !, append(PropAttrs, PropElts, Properties)
        }.
```

3 Predicates

The parser is designed to operate in various environments and therefore provides interfaces at various levels. First we describe the top level defined in `rdf`, simply parsing a RDF-XML file into a list of triples. Please note these are *not* asserted into the database because it is not necessarily the final format the user wishes to reason with and it is not clean how the user wants to deal with multiple RDF documents. Some options are using global URI's in one pool, in Prolog modules or using an additional argument.

load_rdf(+*File, -Triples*)
: Same as `load_rdf`(*File, Triples, []*).

load_rdf(+*File, -Triples, +Options*)

Read the RDF-XML file *File* and return a list of *Triples*. *Options* defines additional processing options. Currently defined options are:

base_uri(*BaseURI*)

If provided local identifiers and identifier-references are globalised using this URI. If omited or the atom `[]`, local identifiers are not tagged.

blank_nodes(*Mode*)

If *Mode* is `share`, blank-node properties (i.e. complex properties without identifier) are reused if they result in exactly the same triple-set. Two descriptions are shared if their intermediate description is the same. This means they should produce the same set of triples in the same order.

expand_foreach(*Boolean*)

If *Boolean* is `true`, expand `rdf:aboutEach` into a set of triples. By default the parser generates `rdf`(*each(Container), Predicate, Subject*).

lang(*Lang*)

Define the initial language (i.e. pretend there is an `xml:lang` declaration in an enclosing element).

ignore_lang(*Bool*)

If `true`, `xml:lang` declarations in the document are ignored. This is mostly for compatibility with older versions of this library that did not support language identifiers.

convert_typed_literal(*:ConvertPred*)

If the parser finds a literal with the `rdf:datatype`=*Type* attribute, call `ConvertPred`(+*Type, +Content, -Literal*). *Content* is the XML element contentas returned by the XML parser (a list). The predicate must unify *Literal* with a Prolog representation of *Content* according to *Type* or throw an exception if the conversion cannot be made.

This option servers two purposes. First of all it can be used to ignore type declarations for backward compatibility of this library. Second it can be used to convert typed literals to a meaningful Prolog representation. E.g. convert '42' to the Prolog integer 42 if the type is `xsd:int` or a related type.

namespaces(*-List*)

Unify *List* with a list of *NS=URL* for each encountered `xmlns`:*NS=URL* declaration found in the source.

entity(+*Name, +Value*)

Overrule entity declaration in file. As it is common practice to declare namespaces using entities in RDF/XML, this option allows for changing the namespace without changing the file. Multiple of these options are allowed.

The *Triples* list is a list of `rdf`(*Subject, Predicate, Object*) triples. *Subject* is either a plain resource (an atom), or one of the terms `each`(*URI*) or `prefix`(*URI*) with the obvious meaning. *Predicate* is either a plain atom for explicitely non-qualified names or a term *NameSpace*:*Name*. If *NameSpace* is the defined RDF name space it is returned as the atom `rdf`. Finally, *Object* is a URI, a *Predicate* or a term of the format `literal`(*Value*) for literal values. *Value* is either a plain atom or a parsed XML term (list of atoms and elements).

3.1 RDF Object representation

The *Object* (3rd) part of a triple can have several different types. If the object is a resource it is returned as either a plain atom or a term *NameSpace*:*Name*. If it is a literal it is returned as `literal`(*Value*), where *Value* takes one of the formats defined below.

- *An atom*
  If the literal *Value* is a plain atom is a literal value not subject to a datatype or `xml:lang` qualifier.
- `lang`*(LanguageID, Atom)*
  If the literal is subject to an `xml:lang` qualifier *LanguageID* specifies the language and *Atom* the actual text.
- *A list*
  If the literal is an XML literal as created by parseType="Literal", the raw output of the XML parser for the content of the element is returned. This content is a list of `element`(*Name, Attributes, Content*) and atoms for CDATA parts as described with the SWI-Prolog SGML/XML parser
- `type`*(Type, StringValue)*
  If the literal has an `rdf:datatype`=*Type* a term of this format is returned.

3.2 Name spaces

XML name spaces are identified using a URI. Unfortunately various URI's are in common use to refer to RDF. The `rdf_parser.pl` module therefore defines the namespace as a `multifile/1` predicate, that can be extended by the user. For example, to parse the Netscape OpenDirectory `structure.rdf` file, the following declarations are used:

```
:- multifile
        rdf_parser:rdf_name_space/1.

rdf_parser:rdf_name_space('http://www.w3.org/TR/RDF/').
rdf_parser:rdf_name_space('http://directory.mozilla.org/rdf').
rdf_parser:rdf_name_space('http://dmoz.org/rdf').
```

The initial definition of this predicate is given below.

```
rdf_name_space('http://www.w3.org/1999/02/22-rdf-syntax-ns#').
rdf_name_space('http://www.w3.org/TR/REC-rdf-syntax').
```

3.3 Low-level access

The above defined `load_rdf/[2,3]` is not always suitable. For example, it cannot deal with documents where the RDF statement is embedded in an XML document. It also cannot deal with really large documents (e.g. the Netscape OpenDirectory project, currently about 90 MBytes), without huge amounts of memory.

For really large documents, the **sgml2pl** parser can be programmed to handle the content of a specific element (i.e. `<rdf:RDF>`) element-by-element. The parsing primitives defined in this section can be used to process these one-by-one.

xml_to_rdf(+*XML, +BaseURI, -Triples*)
Process an XML term produced by `load_structure/3` using the `dialect`(*xmlns*) output option. *XML* is either a complete `<rdf:RDF>` element, a list of RDF-objects (container or description) or a single description of container.

process_rdf(+*Input, :OnTriples, +Options*)

Exploits the call-back interface of **sgml2pl**, calling `OnTriples`(*Triples, File:Line*) with the list of triples resulting from a single top level RDF object for each RDF element in the input as well as the source-location where the description started. *Input* is either a file name or term `stream`(*Stream*). When using a stream all triples are associated to the value of the `base_uri` option. This predicate can be used to process arbitrary large RDF files as the file is processed object-by-object. The example below simply asserts all triples into the database:

```
assert_list([], _).
assert_list([H|T], Source) :-
        assert(H),
        assert_list(T, Source).

?- process_rdf('structure,rdf', assert_list, []).
```

Options are described with `load_rdf/3`. The option `expand_foreach` is not supported as the container may be in a different description. Additional it provides `embedded`:

embedded(*Boolean*)
The predicate `process_rdf/3` processes arbitrary XML documents, only interpreting the content of `rdf:RDF` elements. If this option is `false` (default), it gives a warning on elements that are not processed. The option `embedded`(*true*) can be used to process RDF embedded in *xhtml* without warnings.

4 Writing RDF graphs

The library `rdf_write` provides the inverse of `load_rdf/2` using the predicate `rdf_write_xml/2`. In most cases the RDF parser is used in combination with the Semweb package providing `semweb/rdf_db`. This library defines `rdf_save/2` to save a named RDF graph from the database to a file. This library writes a list of rdf terms to a stream. It has been developed for the SeRQL server which computes an RDF graph that needs to be transmitted in an HTTP request. As we see this as a typical use-case scenario the library only provides writing to a stream.

rdf_write_xml(+*Stream, +Triples*)
Write an RDF/XML document to *Stream* from the list of *Triples*. *Stream* must use one of the following Prolog stream encodings: `ascii`, `iso_latin_1` or `utf8`. Characters that cannot

be represented in the encoding are represented as XML entities. Using ASCII is a good idea for documents that can be represented almost completely in ASCII. For more international documents using UTF-8 creates a more compact document that is easier to read.

```
rdf_write(File, Triples) :-
        open(File, write, Out, [encoding(utf8)]),
        call_cleanup(rdf_write_xml(Out, Triples),
                     close(Out)).
```

5 Testing the RDF translator

A test-suite and driver program are provided by `rdf_test.pl` in the source directory. To run these tests, load this file into Prolog in the distribution directory. The test files are in the directory `suite` and the proper output in `suite/ok`. Predicates provided by `rdf_test.pl`:

suite(+*N*)
: Run test *N* using the file `suite/tN.rdf` and display the RDF source, the intermediate Prolog representation and the resulting triples.

passed(+*N*)
: Process `suite/tN.rdf` and store the resulting triples in `suite/ok/tN.pl` for later validation by `test/0`.

test
: Run all tests and classify the result.

A Metrics

It took three days to write and one to document the Prolog RDF parser. A significant part of the time was spent understanding the RDF specification.

The size of the source (including comments) is given in the table below.

lines	words	bytes	file	function
109	255	2663	rdf.pl	Driver program
312	649	6416	rdf_parser.pl	1-st phase parser
246	752	5852	rdf_triple.pl	2-nd phase parser
126	339	2596	rewrite.pl	rule-compiler
793	1995	17527	total	

We also compared the performance using an RDF-Schema file generated by Protege-2000 and interpreted as RDF. This file contains 162 descriptions in 50 Kbytes, resulting in 599 triples. Environment: Intel Pentium-II/450 with 384 Mbytes memory running SuSE Linux 6.3.

The parser described here requires 0.15 seconds excluding 0.13 seconds Prolog startup time to process this file. The Pro Solutions parser (written in Perl) requires 1.5 seconds exluding 0.25 seconds startup time.

B Installation

B.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 Prolog library files in `$PLBASE/library`, where `$PLBASE` refers to the SWI-Prolog 'home-directory'.

B.2 Windows

Run the file `setup.pl` by double clicking it. This will install the required files into the SWI-Prolog directory and update the library directory.